Diplomarbeit

Automated Generation of JIAC AgentBeans from BPMN Diagrams

Fachbereich *Agententechnologien in betrieblichen Anwendungen und der Telekommunikation (AOT)*

Prof. Dr.-Ing. habil. Sahin Albayrak
Fakultät IV Elektrotechnik und Informatik
Technische Universität Berlin

Vorgelegt von: **Petrus Setiawan Tan**

Betreuer: Dipl.-Inform. Tobias Küster

Petrus Setiawan Tan
Matrikelnummer: 213933
Stettiner Str. 9
10243 Berlin

Die selbstständige und eigenhändige Anfertigung dieser Diplomarbeit versichere ich an Eides statt.

Berlin, 4. November 2011   Petrus Setiawan Tan

Abstract

Models have been used as a solution to bridge the communication gap between the business users and the IT world, a common problem in the software industry. In the domain of multi agent systems, this communication gap is believed to be one of the reason why the agent concept is not very popular in the industry, although it has been a research topic for many years. On the other hand, webservices and service oriented architectures that address a similar problem domain have been adopted much faster by the business users.

At the moment some model driven approach has been made to simplify the development of multi agent systems, providing tools that allow graphical process editing and a mapping of the process into agents. One example is the VSDT which provides BPMN editor, transformation framework and a mapping of BPMN into JIAC agents, thus allowing agents to be designed and created using BPMN. In this work we will extend the VSDT's transformation framework in order to develop a transformation of BPMN to executable Java code or to be more specific, to JIAC Agent Beans.

Zusammenfassung

Die Kommunikationslücke zwischen Unternehmen und IT stellt ein weitverbreitetes Problem in der Softwareindustrie dar. Als Lösung dieses Problems greift man oft auf Modelle zu. Auf dem Gebiet der Multi-Agenten-Systeme wird diese Lücke für eine Ursache gehalten, weshalb das Agentenkonzept, obwohl es seit mehreren Jahren ein Forschungsthema ist, in der Industrie nicht sehr populär ist. Im Gegesatz dazu kommen bei den Unternehmen Konzepte wie Webservices und serviceorientierte Architekturen, die einen ähnlichen Problembereich ansprechen, viel besser an.

Zur Zeit bestehen bereits einige Ansätze die dafür gedacht sind, die Entwicklung von Multi-Agenten-Systeme zu vereinfachen. Diese Ansätze stellen Werkzeuge bereit, die eine graphische Prozessmodellbearbeitung und die Abbildung der Prozesse zu den Agenten ermöglichen. Ein Beispiel dafür ist Visual Service Design Tool (VSDT), das einen BPMN Editor, ein Transformationsframework und eine Abbildung von BPMN auf JIAC Agenten bereitstellt und dadurch die Generierung von Agenten aus BPMN ermöglicht. In dieser Arbeit wird das Transformationsframework des VSDT erweitert, um eine Transformation von BPMN auf Javacode bzw. JIAC Agent Beans zu entwickeln.

Acknowledgement

I would like to thank Tobias Küster for being a very good supervisor, for all the help, suggestions and motivation he gave me during the last 6 months.

I also want to thank my whole family for their constant love and support.

Contents

List of Figures

1. Introduction

In this chapter we will start by introducing the motivation of this work and the problem we would like to solve. Afterwards we will then present the goals of this work and give a short outline about the following chapters.

1.1 Motivation

A common problem in the software engineering is the communication gap between the business people as a client and the IT world. Communicating through models (normally consisting of graphical sketches), especially those which are commonly used in the client's domain such as the BPMN, clearly is a solution in bridging this gap. In the domain of multi agent systems, this communication gap is believed to be one of the reason why the agent concept is not very popular in the business world, although it has been a research topic for many years. On the other hand, webservices and service oriented architectures that address a similar problem domain have been adopted much faster by the business users.

At the moment some model driven approach has been made in order to simplify the development of multi agent systems, providing tools that allow graphical process editing and a mapping of the process into agents. Most of these approaches use a rather simple process model which often solely targets a single agent. Although such approaches can also be used for designing agents, the model has its limits in designing multi agent systems.

The main motivation of this work is to present a solution to this problem, in which the development process of multi agent systems will be simplified. This will help the concepts of software agents and multi agent systems gain more acceptance in the industry. We hope to achieve that by developing a mapping from BPMN to agents that will enable developers to create agents simply by modeling business processes.

1.1.1 Model Driven Engineering

Over the past few years more and more software developers have been adopting the principle of *Model Driven Engineering*(MDE) where they no longer focus on writing

programs but on creating a set of models which defines the software. By modeling the software, the developer creates documents that provide an abstract view of the software system. This should make the designed system easier to understand for non experts e.g. the stakeholders. Moreover, since models are mostly independent from any plattform or a specific programming language, they are applicable in different platforms.

A significant number of the so called Computer Aided Software Engineering (CASE) tools has been developed to support this methodology. Beside providing support in creating and editing the models, most of these CASE tools are also equipped with transformation features that allow us to transform the model into text or even executable programs, thus increasing efficiency in the software development process. We can say that the real benefit of MDE lies in the transformation. By providing a mapping between the model and the code, we can create standardized programs, accelerate development time and minimize faults in writing the code.

1.1.2   MDE in Multi Agent Systems

Back in 2007, an MDE-approach has been made in order to bridge the gap between the industry and the multi-agent systems. As a result, a CASE-tool called the Visual Service Design Tool (VSDT) was developed [14]. VSDT provides the transformation of Business Process Modeling Notation (BPMN) to Business Process Execution Language (BPEL) and Java-based Intelligent Agent Componentware (JIAC) framework. This tool also allows agents to be designed using the very expressive BPMN, which is also a model that has already been manifested in the industry. With this approach, most people from the industry should be able to design software agents.

Another MDE-approach in multi-agent systems was also made by Workflows and Agents Development Environment (WADE) by introducing a workflow which defines an agent's task and can be designed graphically with the developing tool called WOLF. From this workflow a Java code will be generated. However WADE is an example that uses a simple process model which targets only a single agent. We will discuss WADE further in section 3.3.

In the scope of this work, we will present a combination of both approaches and develop a plugin to VSDT to enrich its transformation feature with a code generator that will transform BPMN models into executable Java code, or JIAC Agent Beans to be more specific.

1.2   Goals

As mentioned above, the main goal of this work is to develop an eclipse plugin as an extension to VSDT to enrich its transformation features with a new transformation from BPMN to JIAC Agent Beans.

Having a code generator does not mean that developers are not required to write any code at all. The idea is to generate as much of the code where the model is able to supply with the information required. Because it is nearly impossible to put all implementation details into the model and to anticipate the possibility that the

generated code will be edited manually, considerations have to be made such that conflicts should not occur when the transformation is called upon a code that has been edited manually. So the main challenge of generating executable code is to allow a regeneration of the code (e.g. when the model changes) while protecting the manually added user code.

1.3 Outline

In chapter 2 we will introduce some background topics of this work, starting with JIAC (the target framework), followed by BPMN (the model), VSDT (the framework to be extended) and JET(the technology that is being used).

Chapter 3 will discuss some state of the art topics related to this work such as the transformation from BPMN to BPEL, the existing transformation from BPMN to JADL which is implemented in the VSDT, and WADE, the similar approach in generating a process into java code.

In Chapter 4, the mapping from BPMN to JIAC Agent Beans will be presented with examples showing an excerpt of the process and the associated code mapped from it. We will start with the mapping of pools and processes followed by the mapping of some workflow structures e.g. loops and gateways. Afterwards we will then present the mapping of activities and events.

Chapter 5 contains the details about the transformation's implementation, followed by an example presented in chapter 6.

Finally in chapter 7, the conclusion of this work and some future works related to this topic will be presented.

2. Background

In this chapter we will discuss the some background topics regarding the transformation that should be developed. We will start with Java-based Intelligent Agent Componentware (JIAC), the agent framework which is the target of the transformation followed by Business Process Modeling Notation (BPMN), the modeling notation that is being used. Afterwards we will present the Visual Service Design Tool (VSDT), the existing modeling and transformation framework that should be extended, and finally we will present Java Emitter Templates (JET), the technology that will help us to implement the transformation.

2.1 JIAC

Java-based Intelligent Agent Componentware (JIAC) is a Java-based agent architecture and framework that was developed to simplify the development of software agents[8]. The framework provides features such as FIPA compliant communication, Believe-Desire-Intention (BDI) reasoning, strong migration, web-service connectivity and many others. JIAC also provides high security and advanced accounting mechanisms which makes it suitable for the use in industrial and commercial applications.

JIAC Agent Description Language
With its core component called JIAC Agent Description Language (JADL) [13], JIAC also provides an agent programming language. With JADL, the agent's *plan elements, rules, ontologies* and *services* can be described. JADL is based on a three-predicate-logic that allows the values *true, false and unknown*, making it suitable for open world problems in unknown environments. The actual version of JADL is called *JADL++* [6].

In Figure 2.1 we can see the typical structure of a JIAC application, which consists of Agent Nodes, Agents and Agent Beans. An *Agent Node* is a Java Virtual Machine where the runtime infrastructure for agents, such as discovery services, white and yellow pages services, as well as communication infrastructure, are provided. A
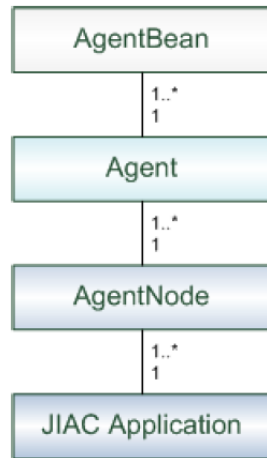
Figure 2.1: JIAC Basic concepts and their structural relationships [4]

JIAC application might be made up of multiple Agent Nodes (distributed application). With the so-called Agent Node Beans, one can extend the Agent Node with additional components.

Each Agent Node may run several *agents*. Agents provide services to other agents and comprise life cycle, execution cycle and a memory. An agent can use infrastructure services in order to find other agents, to communicate with them and to use their services. Skills and abilities of the agent can be extended by the so-called Agent Beans.

2.1.1   Agent Beans

Beside using JADL, another mean to implement the functionality of an agent is by implementing an Agent Bean that is attachable to the agent. In most cases one can implement an Agent Bean simply by extending the class `AbstractAgentBean`. By doing this the new Agent Bean will have access to some useful fields such as :

- `protected Log log` : a logger Instance that can be used to create log messages.

- `protected IAgent thisAgent` : Reference to the agent object that can be used to perform operations on the agent

- `protected IMemory memory` : Reference to the agent's memory that can be used to store and retrieve data.

Agent-Lifecycle
An Agent Bean implements the interface `ILifeCycle` which provides operations to control the bean's state in accordance to the agent's lifecycle (see Figure 2.2) by declaring the methods `init()`, `start()`, `stop()`, and `cleanup()`. These methods are implemented by the class `AbstractLifeCycle`, a super class of `Abstract-AgentBean` which also provides the methods `doInit()`, `doStart()`, `doStop()`, and `doCleanup()`. By overriding these methods we can hook up code that should be executed when the Lifecycle state changes e.g. looking up needed actions, attaching SpaceObeserver to the agent's memory, etc.
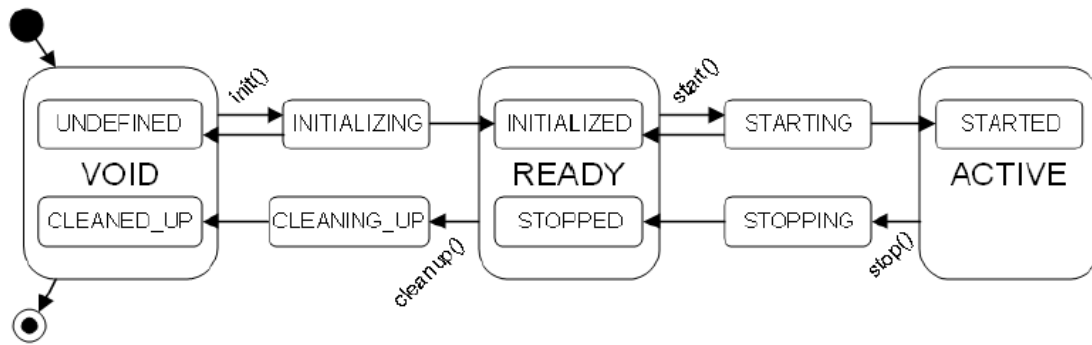
Figure 2.2: Agent-Lifecycle [4]

Execution-Cycle

By implementing the method `execute()`, we can tell the Agent Bean to do something periodically. This method is called by the agent's execution cycle, given that the execution interval property is specified in the configuration file[1], and while the agent is on the **started** state.

Actions

Agent Beans can provide actions. Actions are public methods that can be invoked by another bean and depending on the action's scope it can even be invoked by another agent. Therefore, an action can be seen as a service. Using an action normally includes two or three steps (the third one is optional):

1. find the action

2. invoke the action

3. get the results

The following listing shows an example on how to use the send action provided by the `ICommunicationBean`:

```
1  Action sendAction = retrieveAction(ICommunicationBean.ACTION_SEND); //find action
2  [...]
3  invoke(sendAction, new Serializable{message, receiver})//invoke
```

Listing 2.1: Using an Action

The methods `retrieveAction()` and `invoke()` used in the above listing are provided by the `AbstractAgentBean`.

The simplest way to provide actions is by extending the class `AbstractMethodExposingBean` (also an extension of `AbstractAgentBean`) and put the @Expose Java annotation on the method that we want to expose as an action, as shown in the following listing:

---

[1]Information about the configuration file can be found in the jiac manual[4]

```
1  //static variable for the action name
2  public static final String ACTION_DOACTION = packagename.BeanName#doAction;
3  [...]
4  @Expose(name=ACTION_DOACTION, scope = ActionScope.GLOBAL)
5  public void doAction(){
6  [...]
7  }
```

Listing 2.2: Providing an Action

Memory and SpaceObserver

The default implementation of the agent's memory is a simple tuple space that can
hold any Java objects implementing the `IFact` interface (an extension to java.io.Se-
rializable). The memory provides 4 methods to work on the space: *read, write,
remove* and *update*.

Each time a bean calls an operation that modifies the memory, a SpaceEvent is fired.
There are currently 4 different SpaceEvents:

- `WriteCallEvent` - fired when a new object is written in the memory.

- `UpdateCallEvent` - fired when an object has been updated in memory.

- `RemoveCallEvent` - fired when an object is removed from the memory.

- `RemoveAllCallEvent` - fired when all objects are removed from the memory.

By using SpaceObserver, we can listen to these SpaceEvents and get a notification
whenever a space event is being fired. For example we can implement a space
observer which will start a process when a message is being written in the memory
(listing 2.3).

```
1  SpaceObserver<IFact> observer = new SpaceObserver<IFact>(){
2    public void notify(SpaceEvent<? extends IFact> event){
3      if(event instanceof WriteCallEvent){
4        doProcess(); //start process
5      }
6    }
7  };
8
9  memory.attach(observer);
```

Listing 2.3: A SpaceObserver

To start receiving the SpaceEvents, we should attach the observer into the agent's
memory (see line 9 of listing 2.3). To stop receiving notifications on the SpaceEvents
we can similarly use the `detach()` method.

2.2   BPMN

Business Process Modeling Notation (BPMN) [5] is a standard notation for modeling
business processes, initially published by the Business Process Modeling Initiative
(BPMI) and was later adopted by the Object Management Group (OMG). A busi-
ness process diagram (as seen in figure 2.3) can be compared to UML's activity
diagram.
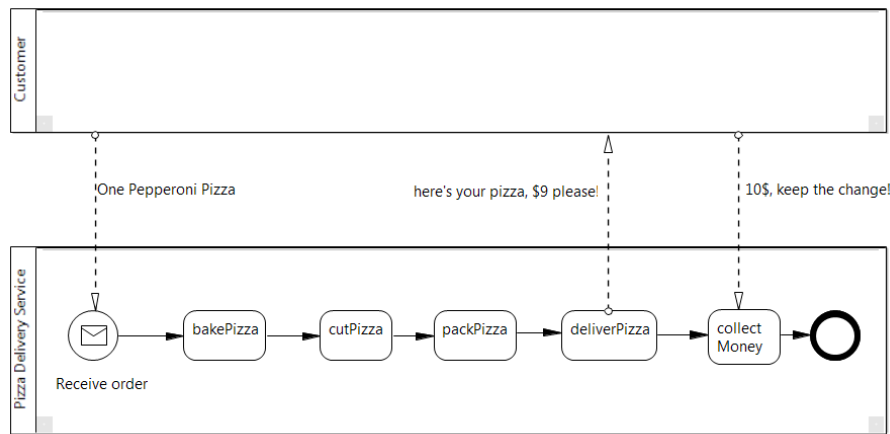
Figure 2.3: A Simple Business Process Diagram

BPMN was made to provide a notation that is understandable by all business users, creating a bridge for the gap between the business process design and the process implementation. Moreover, BPMN shares many concepts e.g. services with multi agent systems. With the mapping of BPMN to agents, we hope to be able to increase the spreading of the multi agent systems in the business world.

2.2.1 Levels of Abstraction

Beside having an easy to read graphical notations, BPMN elements are also equipped with additional attributes (so-called properties), which are hidden from the diagram and provide information needed for automated code generation. The BPMN can be seen as having at least three levels of abstraction:

1. Basic Types - The diagram are made up of easily recognized graphical elements

2. Subtypes - Each basic types can be further distinguished into subtypes with additional graphical elements such as icons that mark the type of an event's trigger.

3. Properties - Each element has additional detailed information that are hidden from the diagram, but needed for the transformation.

2.2.2 BPMN Elements

BPMN elements can be categorized in five basic groups [5]:

- Flow Objects

- Data

- Connecting Objects

- Swimlanes

- Artifacts

Flow Objects include events, activities and gateways. These elements are the most important in BPMN and they are held in a lane.

An *event* describes something that happens during the course of a process. It is divided into start event, intermediate event and end event (see figure 2.4).

start   intermediate   end
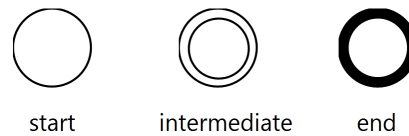
Figure 2.4: BPMN Event Types. From left to right: Start Event, Intermediate Event, End Event

BPMN Events are further divided into subtypes according to the type of the event's trigger (for start and intermediate events) or result (for end events). The event figures (see figure 2.5) are drawn with different icons in the middle, according to the trigger.
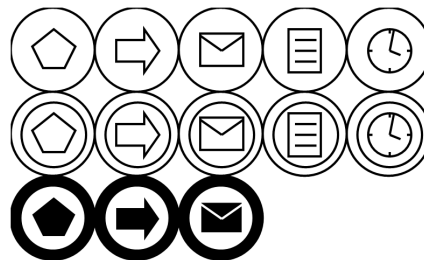
Figure 2.5: Examples of Event Subtypes. From left to right: Multiple, Link, Message, Rule, Timer

An *activity* describes something that is done during a process. It is divided into *tasks* (atomic activities) and *sub processes* (composite activities).

doTask

mySubProcess

doA → doB
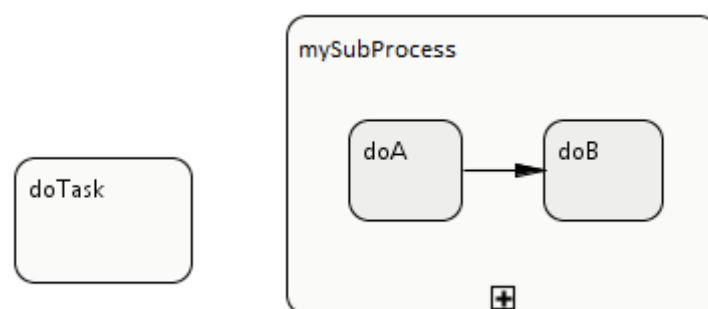
Figure 2.6: Task(Left) and Subprocess

Gateways are used to define all kinds of splitting and merging behavior. Its semantics depends on the dimension of its incoming and outgoing sequence flows. Gateways are divided into the following types (also see figure 2.7):

1. Exclusive

2. Event Based

3. Inclusive

4. Parallel

5. Complex

Figure 2.7: Gateways - From left to right: Exclusive, Event Based, Inclusive, Parallel, Complex

Data is represented by the following four elements:

- Data Objects

- Data Inputs

- Data Outputs

- Data Stores

Data Object describes information that is needed by an activity or what they produce. It can represent a singular object or a collection of objects. *Data Inputs* and *Data Outputs* describe the same information for processes. *Data Stores* describe the location where information, that persists beyond the scope of a process are stored.

With *connecting objects*, flow objects can be connected to each other or to other information. There are 3 different kinds of connecting objects:

- *Sequence Flows* - represent flow control, used for connecting flow objects within a pool in the order of execution.

- *Message Flows* - represent messages being exchanged exclusively between pools.

- *Associations* - mainly used for documentation, e.g. between flow objects and a text annotation.

Swimlanes are divided into *pools* and *lanes*. Each pool represents one participant in the business process, while lanes are used to create partitions within a pool, for example to model different departments of an institution. Figure 2.8 shows a pool with 2 lanes.

Figure 2.8: Pool With 2 Lanes

Artifacts are additional information about the process. They are mainly used for documentation purpose. The current set of artifacts includes:

- Text Annotation

- Group

To provide a rough overview on how the agent technology can support the implementation of business processes, let us take a look on this mapping example of BPMN elements to agents. A pool in a business process diagram can represent an agent, which is able to communicate with other agents (another pool) through messages (represented with the BPMN message flows). Agents can react to events. A detailed mapping needed for implementing the transformation will be discussed later in chapter 4.

2.3 VSDT

The Visual Service Design Tool (VSDT) [9] is a CASE tool developed to support the idea of Process Oriented Agent Engineering [16], where agents are designed by defining use cases and processes described by the BPMN. Its features include the BPMN editor, process structure view, model validation, import of existing web services, transformations to BPEL, and JIAC and many more.

2.3.1 BPMN Editor

The BPMN Editor in VSDT was created using eclipse's Graphical Modeling Framework (GMF). The editor enables us to create and modify business processes graphically. It also provides a properties view, where we can see and edit the additional properties of an element selected in the diagram.
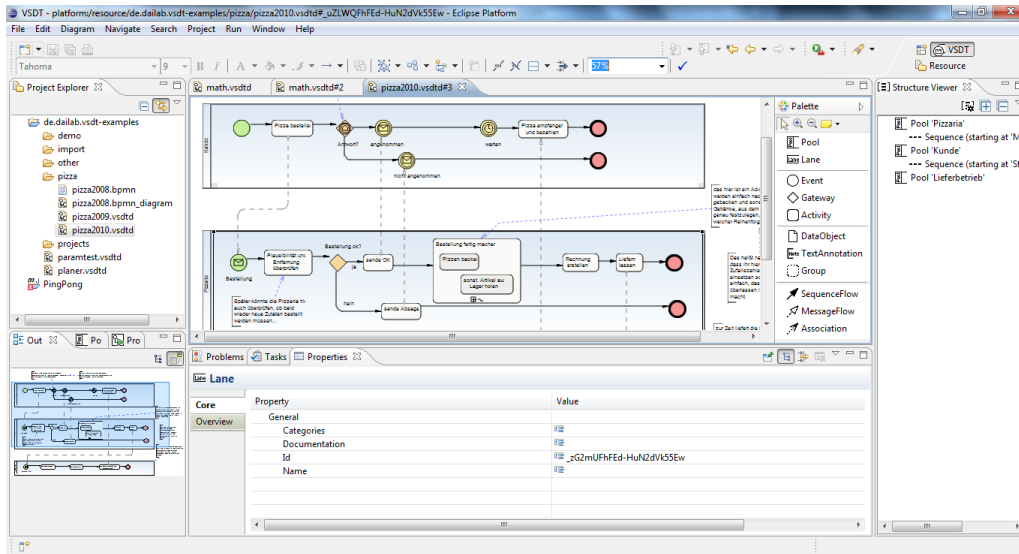
Figure 2.9: VSDT - Editor View

The VSDT was designed to support pure BPMN that is independent from any execution language [15]. This means the standard editor does not support language specific features such as the validation of expression to be conform with the syntax of a specific language (such a validation can be implemented in the transformation). However, the editor can be enriched with plugins e.g one that provides a view with language specific functionality.

2.3.2 Transformation Framework

Since its initial development, VSDT's transformation framework is designed to be extensible and reusable. This makes the development of a new transformation easier. For this purpose the transformation process is subdivided into several stages:

1. *Validation*: Validate the input model.

2. *Normalization*: Prepare the input model for transformation.

3. *Structure Mapping*: Convert the input model to a block-like structure.

4. *Element Mapping*: Perform the actual mapping, create target model.

5. *Clean Up*: Remove redundancies, improve readability, etc.

Due to the fact that the validation, normalization and structure mapping are mostly independent from the target language, the standard mapping provided for these stages are reusable, which makes it possible to implement a new transformation by specifying the element mapping only. Figure 2.10 shows the UML class diagram of the transformation framework with the example transformation to BPEL.
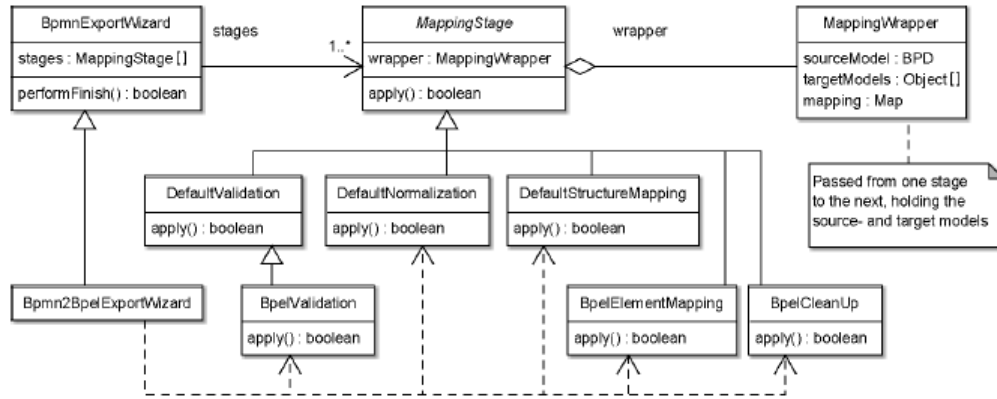
Figure 2.10: Essential classes of the transformation framework, including the BPEL case.[14]

2.4 JET

Java Emitter Templates (JET) [12] is a code generating framework, developed as a part of the Eclipse Modeling Project [7]. Using the so-called templates, one can transform a model into various types of text, from a simple plain text up to text containing HTML, XML or Java code. There is a naming convention for the JET-Template file according to the type of the generated text. For example the name of a template file generating a Java code should end with *.javajet*

The transformation process is done in two steps (see figure 2.4). First, the JET-Builder will translate the template file into a Java class holding a generate method. Then we can create an instance of the template class and call its generate method to get the result String which we can process further e.g. writing it into a file.
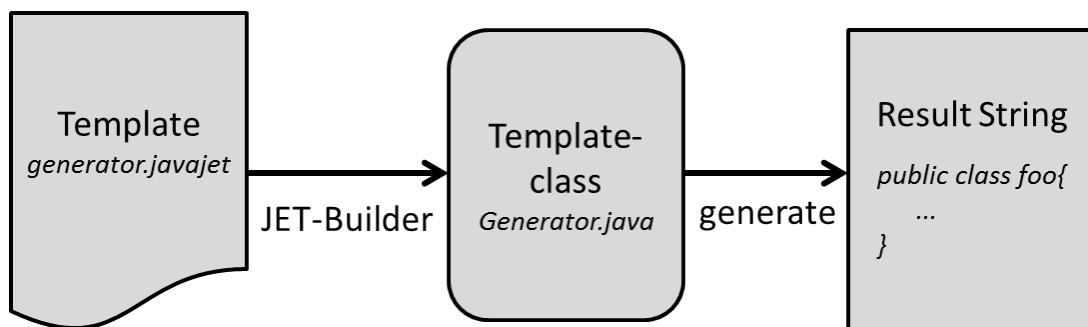
Figure 2.11: JET Transformation Steps.

2.4.1 JET-Templates

JET-Templates use a JSP-like syntax which makes it easy to write and understand. There are three types of expressions in JET-Syntax:

1. **Directives** $<\%@$ *directive* $\{$*attribute* $=$ *"value"* $\}*\%>$
   Directives contain information for the JET-Engine.

2. **Scriptlets** $<\%$ *scriplet* $\%>$
   With Scriptlets we can use any Java code fragment in the template. For example we can type cast the argument object into List:
   `<% List<String> studentlist = (List<String>) argument; %>`
   These code fragments are executed at invocation time of the generate method.

3. **Expressions** $<\%=$ *expression* $\%>$
   An expression contains a Java expression which will be executed at invocation time, and its result will be added into the StringBuffer.

A set of JET templates is called a transformation. It is possible to build this transformation with a main template which acts as a visitor and runs through the model. This main template will then use other templates which handle a specific element of the model. For example, in UML to Java transformation you can have special templates that handle the package, class, variables and methods.

The following listing shows a simple example of a JET-Template that generates an XML:

```
1 <%@ jet package="generator" imports="java.util.*" class="StudentListGenerator" %>
2 <?xml version="1.0" encoding="UTF-8"?>
3 <% List<String> elementList = (List<String>) argument; %>
4 <class>
5     <% for (Iterator i = elementList.iterator(); i.hasNext(); ) { %>
6       <student><%=i.next()%></student>
7     <% } %>
8 </class>
```

Listing 2.4: A Simple JET-Template

A Jet-Template starts with the so-called **jet-directive**. It contains information for the JET-Builder, for example the name of the translated Java class (also called the template class), the package in which the template class should be placed into and a list of classes that should be imported by the template class.

In the first step of the transformation process, the JET-Builder will then translate this template into the Java Class generator.StudentListGenerator:

```
1 package generator;
2 import java.util.*;
3
4 public class StudentListGenerator
5 {
6   protected static String nl;
7   public static synchronized StudentListGenerator create(String lineSeparator)
8   {
9     nl = lineSeparator;
10    StudentListGenerator result = new StudentListGenerator();
11    nl = null;
12    return result;
13  }
14  public final String NL = nl == null ? (System.getProperties().getProperty("line.
         separator")) : nl;
15  protected final String TEXT_1 = " " + NL + "<?xml version=\"1.0\" encoding=\"UTF
         -8\"?>";
16  protected final String TEXT_2 = NL + "<class>" + NL + "\t ";
17  protected final String TEXT_3 = NL + "      <student>";
18  protected final String TEXT_4 = "</student>";
```

```
19   protected final String TEXT_5 = NL + "</class>";
20   public String generate(Object argument)
21   {
22     final StringBuffer stringBuffer = new StringBuffer();
23     stringBuffer.append(TEXT_1);
24      List<String> elementList = (List<String>) argument;
25     stringBuffer.append(TEXT_2);
26      for (Iterator i = elementList.iterator(); i.hasNext(); ) {
27     stringBuffer.append(TEXT_3);
28     stringBuffer.append(i.next());
29     stringBuffer.append(TEXT_4);
30      }
31     stringBuffer.append(TEXT_5);
32     return stringBuffer.toString();
33   }
34 }
```

Listing 2.5: The Translated Java-Class

The most interesting part of the Java template class is the generate method. To get the text that should be generated, an instance of this class should be created and then we pass a list to the generate method, for example with the following code:

```
1   ...
2   List<String> students = new ArrayList<String>();
3   students.add("Peter");
4   students.add("John");
5   students.add("Caroline");
6
7   StudentListGenerator generator = new StudentListGenerator();
8   generator.generate(students);
```

Listing 2.6: An example Java code calling the generate method

, where the result of the transformation will be:

```
1   <class>
2     <student>Peter</student>
3     <student>John</student>
4     <student>Caroline</student>
5   </class>
```

Listing 2.7: Result String of the Transformation

2.4.2   Two different JET-Versions

There are currently two different JET-Versions in the Eclipse Modeling Project. The syntax and examples mentioned above correlate to the older version of JET, which allows us to generate text with an object as argument. In the template, the argument variable can be type-casted into the class of our model. This JET-Version is effective if the model we want to generate the text from is a Java object. We get a String as a result which we can write into a file using the Java-IO or even eclipse API.

In the updated version of JET, also called JET2, some workspace and Java related "tag-libraries" are provided, enabling us to do the transformation without using the Java and Eclipse API. Unfortunately, in this version the model has to be an XML-file. Implementing the transformation of the BPMN directly from its XML-representation will be harder and the template will be confusing. Therefore, a decision has been made to implement the mapping similarly to the existing transformation to JIACV where an intermediate model class of the Agent Bean will be created. This intermediate model will then be transformed into Java code using the older version of JET. More details on the implementation will be discussed in chapter 5.

2.4.3 New JET2 Elements

The enhanced version of JET comes with some new syntax elements such as:

- Comments <-- *Comment* -->
  Comments will be copied to the translated Java class as a line comment, but they will have no influence on the transformation.

- Java Declaration <%! ... %>
  Within the Java declaration tag, the template may declare some Java methods or variables.

- The @taglib directive <%@taglib id="..." prefix="..."%>
  With the @taglib directive, we can import an XML Tag Library or rename the tag library's namespace prefix.

- Custom XML tag
  JET2 templates may contain custom XML tags. JET2 comes with four standard tag libraries, and we can also define our own tag library.

Out of the new elements mentioned above, the custom XML tag is the most interesting since JET2 comes with four helpful standard tag libraries[20]:

1. Control tags
   Contain control flow and data management tags. These tags will help us in visiting the input model, evaluate expression, perform loops, iterate through a certain set of elements and read or write some information in the input.

2. Workspace tags
   Contain a set of library that performs operations against the eclipse workspace (for example for creating a file in an existing eclipse project).

3. Java tags
   Contain tags that are useful for generating Java code and can be used for creating Java packages and classes, managing imports etc.

4. Format tags
   Standard tags for formatting text.

3. State of the Art

3.1   BPMN to BPEL Transformation

In the BPMN specification, the mapping from BPMN to BPEL has been included since the first version. In fac,t the development of BPMN was driven by the lack of standard notation for the WS-BPEL[22]. The limitation of the mapping from BPMN to BPEL has also been discussed in various papers, focusing the issues resulted from the incompatibility of the graph structured BPMN and the block structured BPEL, and led to some more sophisticated mapping such as the one introduced by Ouyang et al. ([17],[18]).

At the moment, there are some tools that support the transformation from BPMN to BPEL. In the VSDT, a mapping from BPMN to BPEL based on the mapping given in the BPMN specification was also developed [14], covering nearly every mapping given in the specification including event handlers, inclusive OR and event based XOR Gateways.

For this project the mapping from BPMN to BPEL given in the BPMN specification is used to gain a better understanding of the notation's semantics.

3.2   Existing Transformation to JIAC

At the moment the VSDT is already equipped with a transformation of BPMN to JIAC (the successor of JIAC) which generates JADL files. As mentioned in chapter 2, both JADL and Agent Bean are tools to implement the functionality of an agent in JIAC. The transformation to JIAC Agent Beans is in no way a replacement to the existing transformation. Instead both transformations shall complement each other as the products of both transformations have their own advantages, some of which are listed in the following table 3.1

Advantages of:	
JADL	Agent Beans
can be deployed to a running agent	Written fully in Java, therefore developer friendly. Java is more expressive than JADL. Better performance because no parser is involved.

Table 3.1: Advantages of JADL and Agent Beans

Because JADL files can be deployed to a running agent, the mapping to JADL is suitable for creating dynamic behaviors and services that might be changed and deployed at runtime. On the other hand, the mapping to JIAC Agent Beans is a better choice for creating the core components of an agent because of its expressiveness and better performance.

3.3 Workflows and Agents Development Environment

A similar approach (designing agents behavior with processes and transforming it into Java Code) has been developed by the Telecom Italia with their Java Agent Development Framework (JADE) extension called Workflows and Agents Development Environment (WADE). While JADE was developed to simplify the implementation of software agents, WADE extended JADE with a workflow engine, making it possible to create agents that execute tasks defined as workflows.

3.3.1 Java Agent Development Framework

JADE [1, 2] is an application framework and a middleware written in Java, which supports the development of software agents. The framework provides distributed runtime environments, agent and behavior abstractions as well as communications between agents and discovery mechanisms. We can say that its role is very similar to JIAC.

3.3.2 Workflows and Agents Development Environment

WADE [10] is an extension to JADE, that enriches the application framework with a workflow engine. The WorkflowEngineAgent extends the JADE basic agent class with an ability to execute workflows represented in a WADE specific formalism.

A WADE workflow is actually a Java class, thus it can be edited and managed as Java classes and can contain pieces of code which are needed to implement the process. With WOLF, a development environment that comes with WADE, developers can edit the workflow graphically as well as textually. The code view and the graphical view of the workflow are kept in sync.

Despite having all the advantages of a Java code, the WADE workflow is rather simple and not so expressive as the BPMN. Similarities to the agent technology such as event handling and communication flow are also missing in the workflow. Further because each workflow model is associated to a single Java class, it targets a single agent and is not very suitable in designing multi agent systems. In the following we will take a closer look on the workflow concept of WADE.

3.3.3 Workflow Metamodel - Graphical View

For the graphical view of the workflow, WADE adopts a workflow metamodel quite similar to that defined in the XML Process Definition Language (XPDL) standard of the Workflow Management Consortium (www.wfmc.org). In figure 3.1 (image taken from [3]), we can see an example process summarizing the main elements of the WADE metamodel.
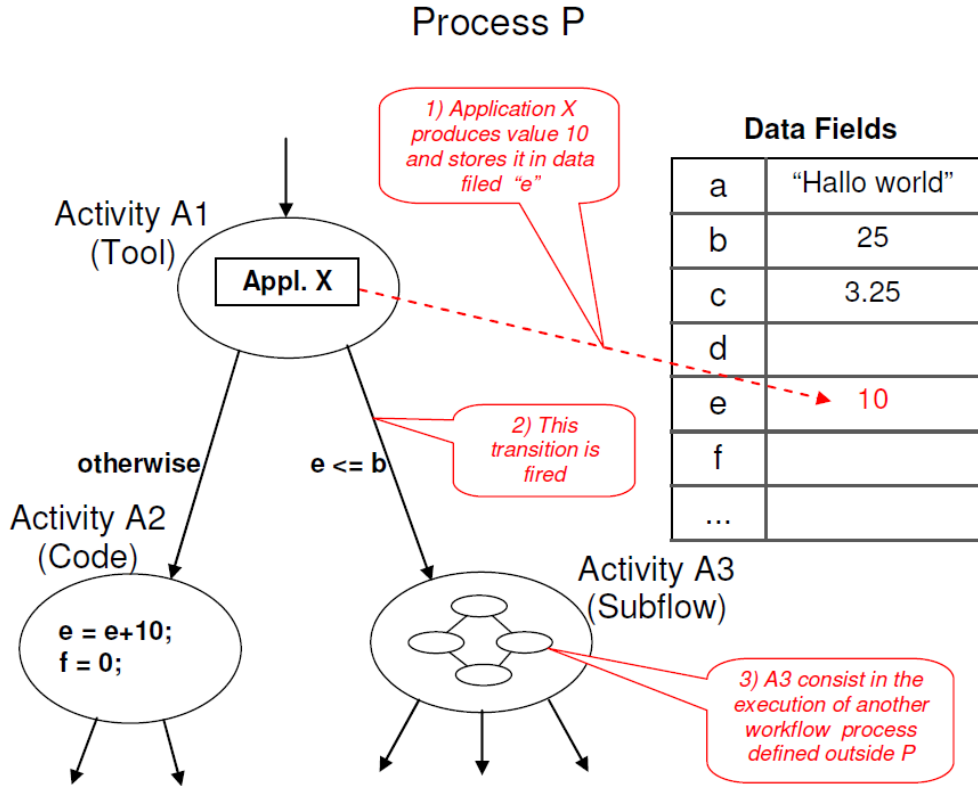
Figure 3.1: Elements of WADE Metamodel [3]

A Process in WADE is made up of a set of activities, where each activity corresponds to the execution of the given operations. A process is defined by exactly one **Start Activity**, and one or multiple **End Activities**. A process may have formal parameters, which define the type of required inputs and expected outputs.

Activities in WADE are divided into different types depending on the operations included in the activity. Six types are mentioned in [21] for being the most relevant:

- ***Tool Activities***
  Contain the invocation of one or more **Applications**, computational entities defined outside of the workflow process and wrapped by a uniform interface.

- ***Subflow Activities***
  Contain the invocation of another workflow process. The execution of the subflow takes place in a different computational place and it may be carried out by another agent. Further we can set whether the subflow should be executed synchronously or asynchronously.

- **Webservice Activities**
  Contain the invocation of a webservice.

- **Code Activities**
  Included operations are specified directly by a set of Java code.

- **Subflow Join**
  Contain operations that will block the main workflow process and wait until the previously launched asynchronous subflow completes, and get the results.

- **Route Activities**
  Route activities are empty. No operation is included. It can be used to simplify complex flows.

Each non ending activity has one or more outgoing **transitions** leading to another activity. A transition may be associated with a condition. Once an activity completes, the conditions of all outgoing transitions will be checked. If the condition holds, the transition will be fired and the workflow continues with the execution of the destination activity.

A process has a set of **data fields** which can be referenced anywhere in the process e.g. in the condition of a transition or in the operations included in an activity.

3.3.4   Workflow Implementation (Code View)

As mentioned before, workflow in WADE is actually a (well structured) Java class. A workflow process is implemented by a Java class extending the `WorkflowBehaviour` class, which provides the methods `registerActivity()` and `registerTransition()` for adding activities and transitions into the process.

The `registerActivity()` method takes a behavior object as argument. There are different behavior classes corresponding to the different activity types discussed in the previous subsection. The actual operations included for the registered activity are wrapped in a void method of the workflow class. The method's name is derived from the activity's name added with the "execute" prefix e.g `executecheckBalance()` for the activity "checkBalance".

The `registerTransition()` method takes a transition object as an argument. If the transition is associated with a condition, then a boolean method (with the prefix "check" added to the condition name) will be created.

In figure 3.2 we can see an example process in its graphical view and listing 3.1 shows the equivalent code view to the example.

Account Withdrawal Process
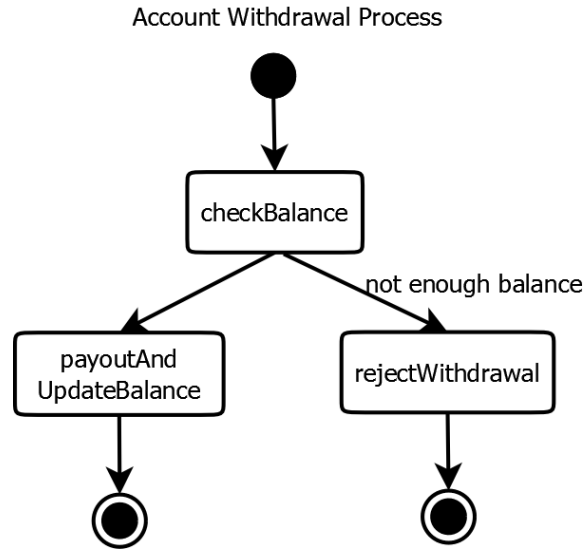
Figure 3.2: WADE Example: Account Withdrawal Process (graphical view)

```
1  //the layout information of the graphical view
2  @WorkflowLayout(entryPoint = @MarkerLayout(position = "(260,31)", activityName = "
       checkBalance"), exitPoints = { }, transitions = {@TransitionLayout(to = "
       rejectWithdrawal", from = "checkBalance"), @TransitionLayout(to = "
       payoutAndUpdateBalance", from = "checkBalance") }, activities = {
       @ActivityLayout(position = "(357,175)", name = "rejectWithdrawal"),
       @ActivityLayout(position = "(116,171)", name = "payoutAndUpdateBalance"),
       @ActivityLayout(position = "(222,78)", name = "checkBalance") })
3
4  public class AccountWithdrawal extends Workflowbehavior {
5    //Data fields
6    public static final String NOTENOUGHBALANCE_CONDITION = "notEnoughBalance";
7    public static final String REJECTWITHDRAWAL_ACTIVITY = "rejectWithdrawal";
8    public static final String PAYOUTANDUPDATEBALANCE_ACTIVITY = "
         payoutAndUpdateBalance";
9    public static final String CHECKBALANCE_ACTIVITY = "checkBalance";
10   private double balance;
11
12   //formal Parameter
13   @FormalParameter(mode=FormalParameter.INPUT)
14   public double amountToBeWithdrawn;
15
16   // All activities should be registered in this method
17   private void defineActivities() {
18     CodeExecutionbehavior checkBalanceActivity = new CodeExecutionbehavior(
19         CHECKBALANCE_ACTIVITY, this);
20     registerActivity(checkBalanceActivity, INITIAL);
21     CodeExecutionbehavior payoutAndUpdateBalanceActivity = new
           CodeExecutionbehavior(
22         PAYOUTANDUPDATEBALANCE_ACTIVITY, this);
23     registerActivity(payoutAndUpdateBalanceActivity, FINAL);
24     CodeExecutionbehavior rejectWithdrawalActivity = new CodeExecutionbehavior(
25         REJECTWITHDRAWAL_ACTIVITY, this);
26     registerActivity(rejectWithdrawalActivity, FINAL);
27   }
28
29   // All transitions should be registered here
30   private void defineTransitions() {
31     registerTransition(new Transition(), CHECKBALANCE_ACTIVITY,
32         PAYOUTANDUPDATEBALANCE_ACTIVITY);
33     registerTransition(new Transition(NOTENOUGHBALANCE_CONDITION, this),
34         CHECKBALANCE_ACTIVITY, REJECTWITHDRAWAL_ACTIVITY);
35   }
36
37   //activity methods
38   protected void executecheckBalance() throws Exception {
39     [code for checking the account's balance]
```

```
40    }
41
42    protected void executepayoutAndUpdateBalance() throws Exception {
43      [...]
44    }
45
46    protected void executerejectWithdrawal() throws Exception {
47      [...]
48    }
49
50    //check method for the condition of a transition
51    protected boolean checknotEnoughBalance() throws Exception {
52      return true;
53    }
54
55 }
```

Listing 3.1: WADE Example: Account Withdrawal Process (code view)

From the above listing, we can see the structure of a workflow implementation class in WADE. At line 3 we can see how the layout information for the graphical view is coded with the Java annotation mechanism. Data fields are implemented as a class variable (line 6-10) so that it can be referenced everywhere in the workflow process.

Although it is not strictly necessary, all activities and transitions should be registered in the methods `defineActivities()` and `defineTransitions()` because the graphical editor will search these methods to detect the activities and transitions to show.

In line 38 to 47 we can see the activity methods that wrap the operations included in an activity. And finally we can see the boolean method `checknotEnoughBalance()` that checks for the condition of the transition from checkBalance to rejectWithdrawal. These methods are per default empty, leaving the operations to be added directly into the code by the developer.

WADE's approach in implementing the workflow as Java code has inspired us for this project. Some of the concepts e.g wrapping the activity in a Java method can also be found in our BPMN to Agent Bean transformation.

4. Mapping BPMN to JIAC Agent Beans

The element mapping from BPMN to JIAC Agent Beans is created based on the existing mapping to JADL. Compared to a JADL script, an Agent Bean that is written completely in Java enables more possibilities in mapping concepts such as intermediate event handling. Some concepts found in WADE's workflow implementation inspired us for the development of the mapping. This chapter will provide a detailed overview of the mapping.

4.1 Pools and Processes

Every pool in a process diagram will be mapped into an Agent Bean. The name of the Agent Bean will be derived from the pool's name and the name of the process diagram it is contained in. If a pool with the same name (e.g Mathematican) is contained in the business process diagrams ExtractRoot and Faculty, then the two agent beans `Mathematican_ExtractRoot` and `Mathematican_Faculty` will be created. Because they were both generated from the pool Mathematican, they are grouped in the package mathematican.

The process contained in a pool is mapped into a **workflow method**. Depending on the start event's type, this workflow method may be called in the execute method of the bean (for timer start event) or it may be exposed as an action (for message start event with service implementation). If the start event is a message start event with MessageChannel implementation, a SpaceObserver will be created and attached to the agent's memory. This SpaceObserver will call the workflow method when a message with the payload and address as specified in the implementing MessageChannel is written into the agent's memory.

The mapping of a task contained in a process flow will be wrapped in the so-called **activity method**. Activity methods will be called by the workflow method. Special cases such as tasks with event handler and subprocesses will be discussed in a later section.

Lanes are currently ignored. A process of a lane will be handled as a process of the containing pool.
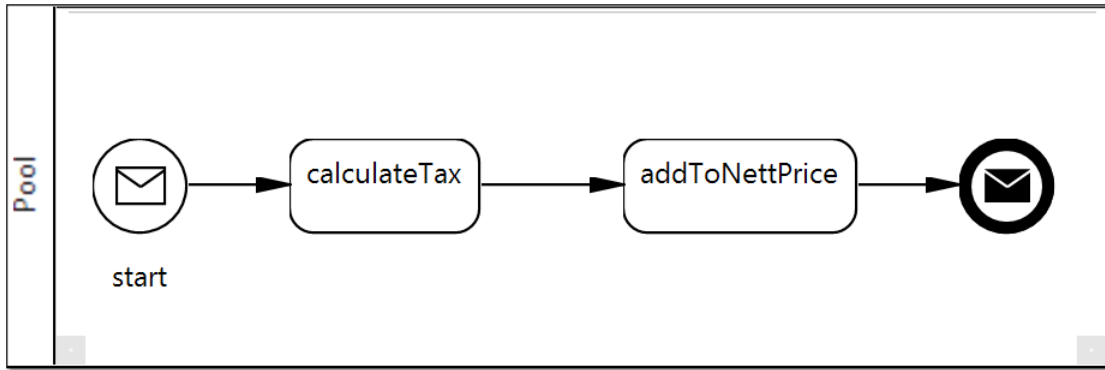
Figure 4.1: Mapping Example : Pool and Process

The figure 4.1 shows a simple example of a pool called Pool with the process do-Process. For the pool, a Java class called Pool_DoProcess will be created and it will have the workflow method called doProcess. In this example the message start event is implemented as a service, thus the workflow method is exposed as an action as we can see in the following listing 4.1.

```java
package pool;//derived from the pool name

//some imports...

public class Accounting_CalculatePrice extends AbstractMethodExposingBean{

  public final static String ACTION_DOPROCESS = "pool.Pool_DoProcess#doProcess";

  // process attribute would be declared here
  [...]
  @Expose(name = ACTION_DOPROCESS, scope = ActionScope.GLOBAL)
  public double doProcess(double nettPrice, double taxRateInPercent) {
    calculateTax();
    addToNettPrice();
    return totalPrice;
  }

  private void calculateTax(){
    [...]
  }

  private void addToNettPrice(){
    [...]
  }
}
```

Listing 4.1: Mapped Element: Pool and Process (Figure 4.1)

4.2 Workflow Structure

Workflow structure is mapped to the content of the workflow method. It defines the invocation structure of the flow objects contained in a process.

4.2.1 Sequence Flow

The mapping of a sequence flow is trivial. The mapped elements connected with a sequence flow will be invoked sequentially in the workflow method (see figure 4.2).

```
1  doA();
2  doB();
3  [...]
4  doC();
```
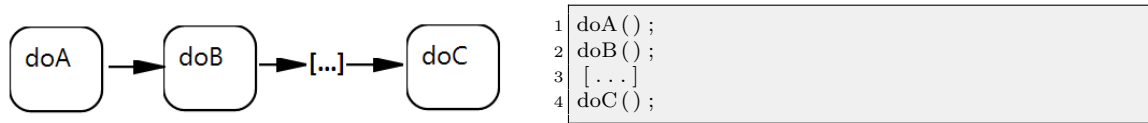
Figure 4.2: Mapping Example: Sequence Flow

4.2.2 Gateways

Branches of a gateway are wrapped according to the gateway's type. There are five different types of gateways:

1. AND (Parallel)

2. OR (Inclusive)

3. XOR_Data (Exclusive)

4. XOR_Event (Event Based)

5. Complex

AND-Gateway (Parallel)
In an AND-Gateway, all branch will be wrapped in parallel to one another. At runtime all branches are executed within a thread. Figure 4.3 shows an example of the mapping.


```
1  [...]
2  Thread and_branch1 = new Thread(){
3    public void run(){
4      doA();
5    }
6  };
7  [create thread for other branches]
8  Thread and_branch2 = new Thread(){
9    public void run(){
10     doB();
11   }
12 };
13 and_branch1.start();
14 [start other branches]
15 and_branch2.start();
16 try {
17   and_branch1.join();
18   [join other branches]
19   and_branch2.join();
20 } catch(InterruptedException) { }
21 [...]
```
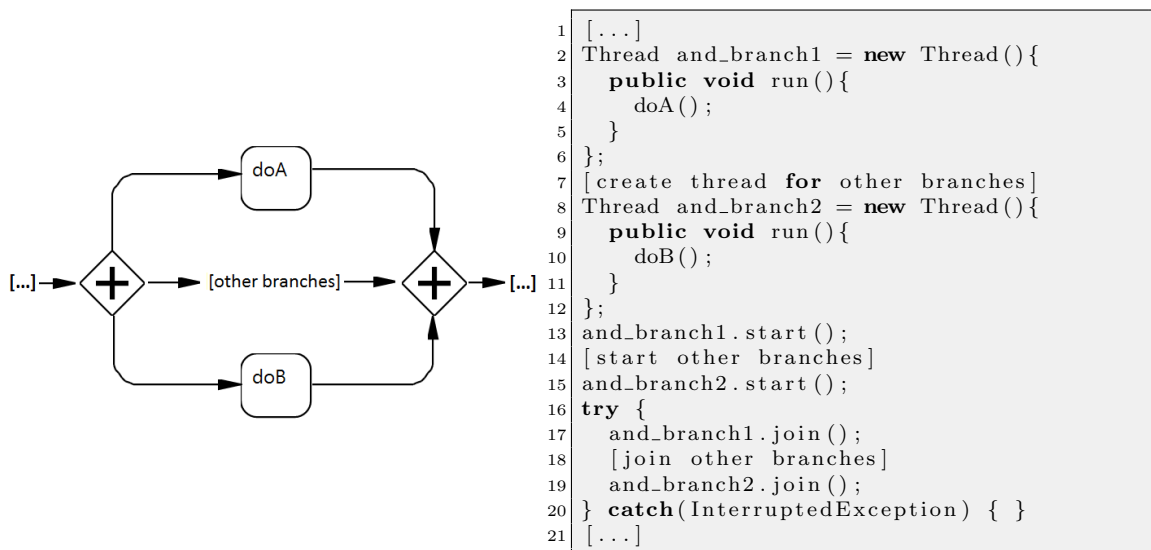
Figure 4.3: Mapping Example: AND Gateway

OR-Gateway (Inclusive)
Branches of an OR-Gateway will also be executed in parallel to one another, but the content of a branch is additionally wrapped in an if-then block as shown in figure 4.4. At runtime, the condition of each branches will be checked and the branch will be skipped if the condition does not hold. However, if the OR-Gateway has a branch

```
1  [...]
2  Thread or_branch1 = new Thread(){
3     public void run(){
4        if(cond1){
5           doA();
6        }
7     }
8  };
9  [thread for other branches]
10 Thread or_branch2 = new Thread(){
11    public void run(){
12       if(cond2){
13          doB();
14       }
15    }
16 };
17 or_branch1.start();
18 [start other branches]
19 or_branch2.start();
20 try {
21    or_branch1.join();
22    [join other branches]
23    or_branch2.join();
24 } catch(InterruptedException) { }
25 [...]
```
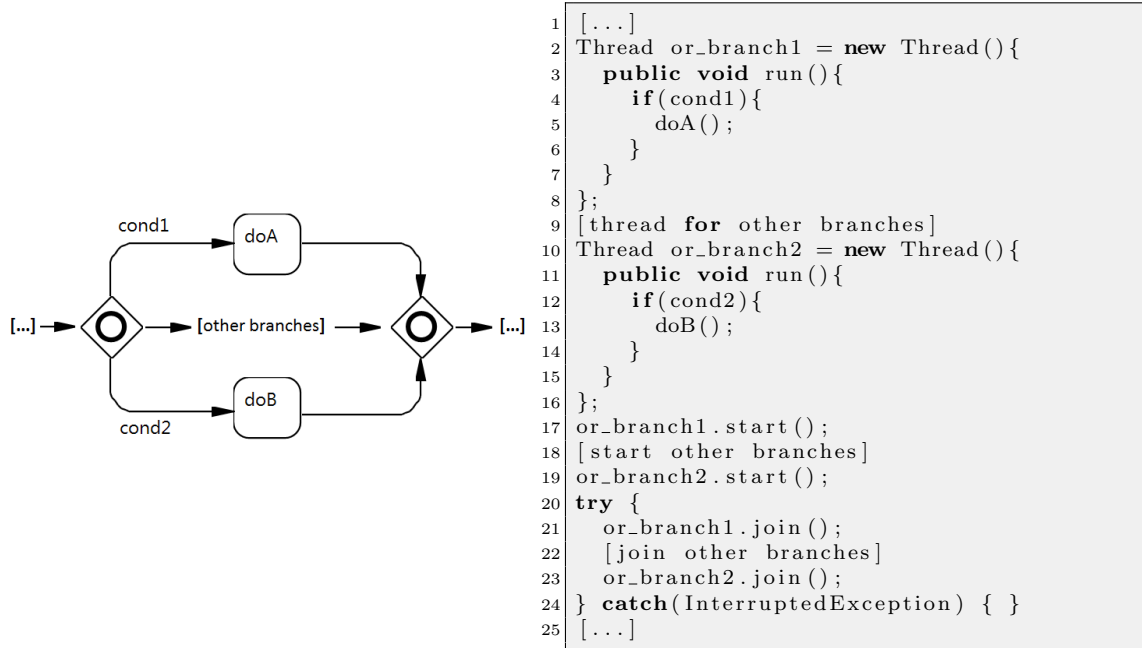
Figure 4.4: Mapping Example: OR Gateway

with default condition, the default branch will only be executed if none of the other branches is executed.

XOR_Data-Gateway (Exclusive)
Branches of an XOR_Data are wrapped in an if-then-else block (see figure 4.5). Only one branch will actually be executed at runtime.

```
1  [...]
2  if(cond1){
3     doA();
4  }else{
5     if([other condition]){
6        [other branches]
7     }else{
8        if(cond2){
9           doB();
10       }
11    }
12 [...]
```
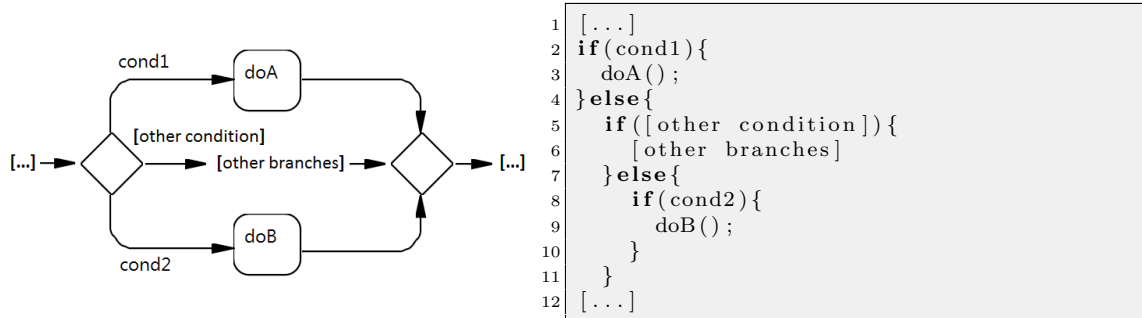
Figure 4.5: Mapping Example: XOR_Data Gateway

XOR_Event-Gateway (Event Based)
For XOR_Event, a waiting loop will be started in a thread, and an EventHandler (an extension to java.lang.Thread) instance for each branch will be created and started, according to the event's trigger of each branch. If an event handler receive an event, the waiting thread will be stopped and the process continues with the elements of the branch. Other branches will be skipped. Figure 4.6 shows an example of the mapping.

The event handler class will be included as an inner class of the bean. We will present the event handler class later in section 4.2.4.

```
1  [...]
2  Thread xor_event_wait = new Thread(){
3      public void run() {
4          while(true) {
5              try {
6                  Thread.sleep(100);
7              } catch(InterruptedException e) {}
8          }
9      }
10 };
11 MessageEventHandler messageHandler = new
        MessageEventHandler(xor_event_wait);
12 [handler for other branches]
13 TimeoutEventHandler timeoutHandler = new
        TimeoutEventHandler(20000,
        xor_event_wait);
14 //start the wait activity and all
        handlers
15 xor_event_wait.start();
16 messageHandler.start();
17 [start other handlers]
18 timeoutHandler.start();
19 try{
20     xor_event_wait.join();
21     timeoutHandler.stop();
22     [stop other handlers]
23     messageHandler.stop();
24 } catch (InterruptedException e) {
25     if(messageHandler.hasBeenTriggered()){
26         doA();
27     }
28     [other branches]
29     if(timeoutHandler.hasBeenTriggered()){
30         doB();
31     }
32 }
33 [...]
```

Figure 4.6: Mapping Example: XOR_Event Gateway

Complex-Gateway
A mapping concept for Complex Gateways has not been developed in the current
version.

4.2.3 Loop Blocks

Structured loop blocks are mapped as shown in figure 4.7.

```
1  [...]
2
3  while(cond){
4      doA();
5  }
6
7  [...]
```

Figure 4.7: Mapping Example: Loop Blocks

While the condition applies, the content branch will be repeated.

4.2.4   Event Handler

As we can see in figure 4.8, the mapping of an event handler attached to an activity
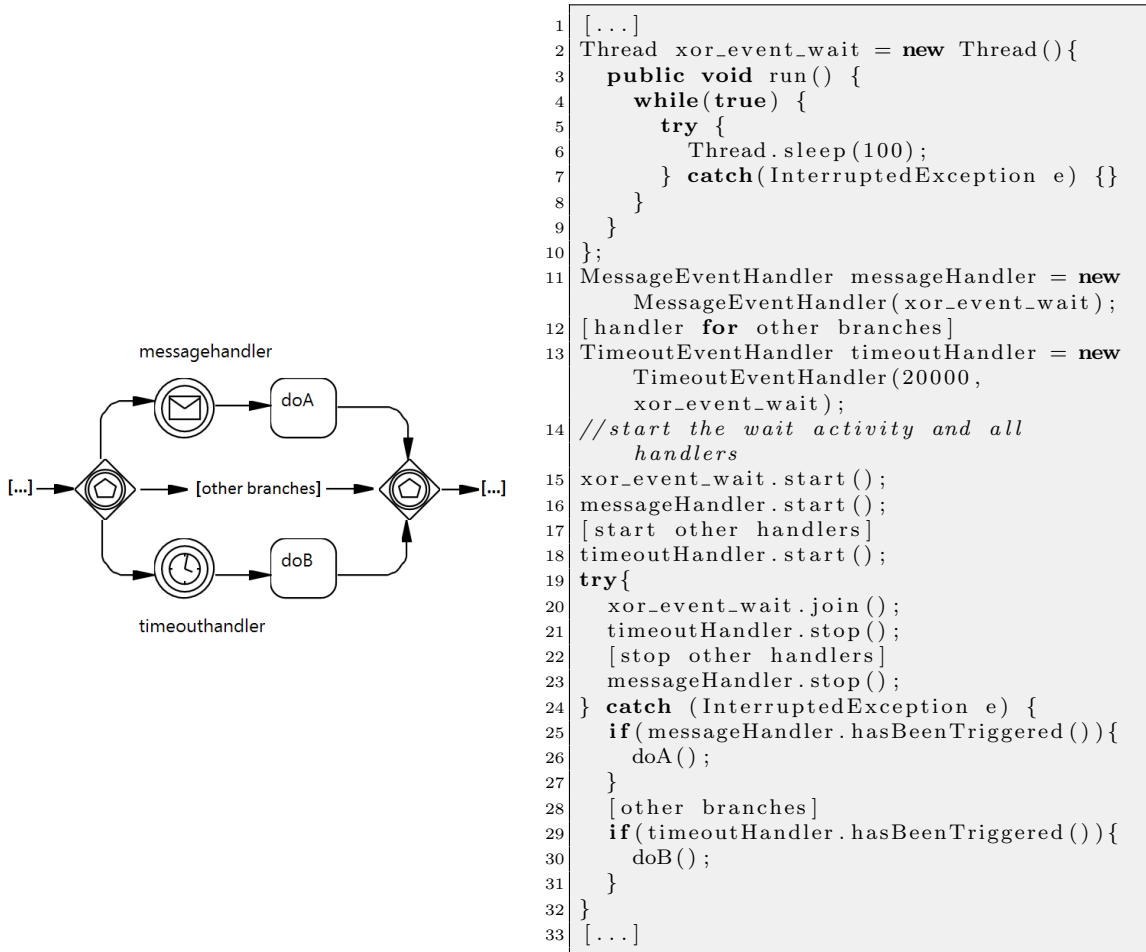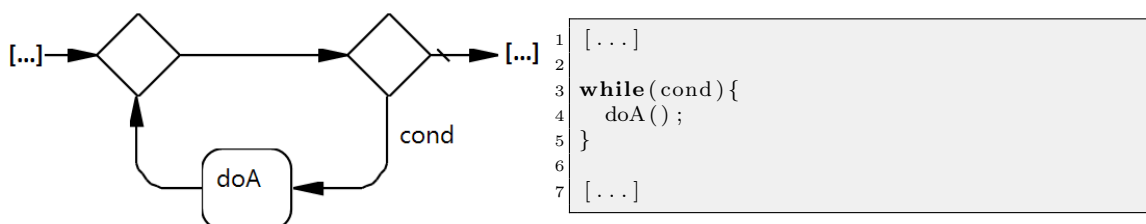is somewhat similar to the mapping of event based gateway. Instead of the waiting
thread, a thread calling the activity method will be created, and it will be stopped
if the event handler is triggered.


```
1  [...]
2  Thread doA_Thread = new Thread(){
3     public void run(){
4        doA();
5     }
6  };
7  MessageEventHandler messageHandler = new
         MessageEventHandler(doA_Thread);
8  TimeoutEventHandler timeoutHandler = new
         TimeoutEventHandler(20000, doA_Thread);
9  [other handlers]
10
11 doA_Thread.start();
12 messageHandler.start();
13 timeoutHandler.start();
14 [start other handlers]
15
16 try{
17    doA_Thread.join();
18    timeoutHandler.stop();
19    messageHandler.stop();
20    [stop other handlers]
21 } catch(InterruptedException e) {
22    if(messageHandler.hasBeenTriggered(){
23       doC();
24    }
25    if(timeoutHandler.hasBeenTriggered(){
26       doD();
27    }
28    if([other handlers].hasBeenTriggered()){
29       [...]
30    }
31 }
32
33 //doB only if none of the
34 //event handlers are triggered
35 if(!messageHandler.hasBeenTriggered()){
36    if(!timeoutHandler.hasBeenTriggered()){
37       if(![other handlers].hasBeenTriggered()){
38          doB();
39       }
40    }
41 }
42
43 [...]
```
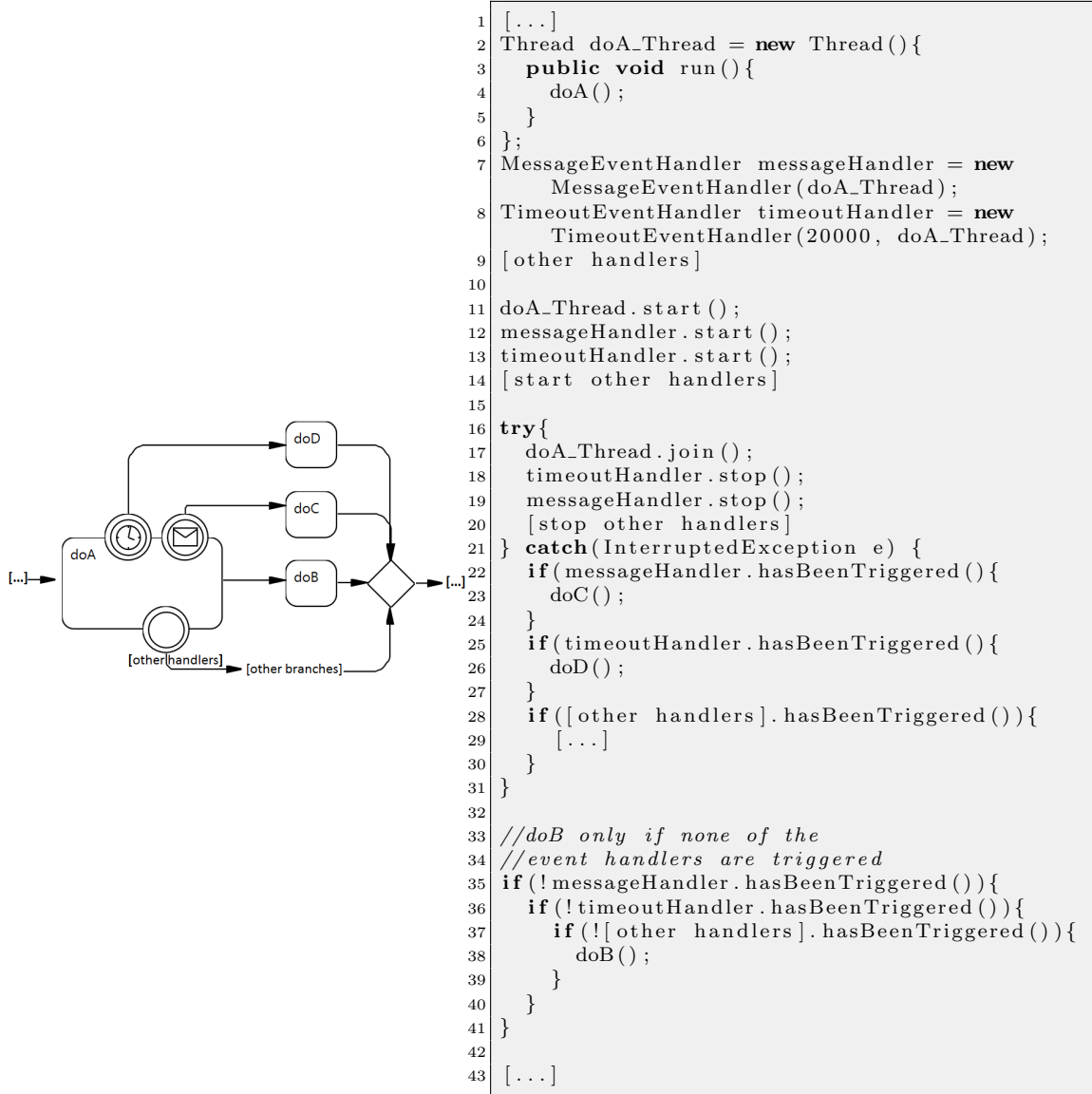
Figure 4.8: Mapping Example: Event Handler

Like we mentioned earlier in the mapping of event based gateway, the event handler
will be mapped into an inner class of the Agent Bean. Currently we have three
different event handler classes:

1. MessageEventHandler - handles message events.

2. TimeoutEventHandler - handles time event where the given time expression is
   a duration.

3. TimeEventHandler - handles time event where the given time expression is an exact time.

Each event handler class has a constructor with a `Thread toStop` argument that will be stopped if the expected event is received. Further they will also have a boolean method `hasBeenTrigerred()` to check whether the event handler has been activated by the expected event. The following code listings show the generated code of the event handlers:

```
class MessageEventHandler extends Thread{
    Thread toStop;
    boolean triggered = false;
    String address;
    Class payloadClass;
    SpaceObserver<IFact> observer;
    Action joinAction;
    Action leaveAction;
    IGroupAddress groupAddress;

    public MessageEventHandler(String channel, String payloadType, Thread toStop){
        address = channel;
        this.toStop = toStop;
        joinAction = retrieveAction(ICommunicationBean.ACTION_JOIN_GROUP);
        leaveAction = retrieveAction(ICommunicationBean.ACTION_LEAVE_GROUP);
        groupAddress = CommunicationAddressFactory.createGroupAddress(address);
        try {
            payloadClass = ClassLoader.getSystemClassLoader().loadClass(payloadType);
        } catch (ClassNotFoundException e) {
            log.error("Class "+payloadType+" not Found!");
            e.printStackTrace();
        }
        observer = new SpaceObserver<IFact>(){
            public void notify(SpaceEvent<? extends IFact> event) {
                if(event instanceof WriteCallEvent<?>){
                    Object obj = ((WriteCallEvent) event).getObject();
                    if(obj instanceof IJiacMessage){
                        IJiacMessage msg = (IJiacMessage)obj;
                        if(msg.getHeader(IJiacMessage.Header.SEND_TO).equals(address) &&
                            payloadClass.isInstance(msg.getPayload())){
                            memory.remove(msg);
                            compensate();
                        }
                    }
                }
            }
        };
    }

    public void run(){
        invoke(joinAction, new Serializable[]{groupAddress});
        memory.attach(observer);
    }

    public void compensate(){
        triggered = true;
        detach();
    }

    public void detach(){
        memory.detach(observer);
        invoke(leaveAction, new Serializable[]{groupAddress});
    }

    public boolean hasBeenTriggered(){
        return triggered;
    }
}
```

Listing 4.2: MessageEventHandler Implementation

```java
class TimeoutEventHandler extends Thread{
    long timeout;
    Thread toStop;
    boolean triggered = false;

    public TimeoutEventHandler(long timeout, Thread toStop){
      this.timeout = timeout;
      this.toStop = toStop;
    }

    public void run(){
      try {
        Thread.sleep(timeout);
        triggered = true;
        toStop.stop();
      }catch(InterruptedException e ) { }
    }

    public boolean hasBeenTriggered(){
      return triggered;
    }
}
```

Listing 4.3: TimeoutEventHandler Implementation

```java
class TimeEventHandler extends Thread{
    Date whenToStop;
    Thread toStop;
    boolean triggered = false;

    public TimeEventHandler(String timeExpression, Thread toStop){
      try {
        whenToStop = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSSZ").parse(
            timeExpression);
      } catch (ParseException e) {
        System.out.println("ParseException: Time Expression has to be in yyyy-MM-dd
            'T'HH:mm:ss.SSSZ format!");
        e.printStackTrace();
      }
      this.toStop = toStop;
    }

    public void run(){
      long time = whenToStop.getTime();
      while(time>System.currentTimeMillis()){
        try {
          Thread.sleep(100);
        }catch(InterruptedException e ) { }
      }
      toStop.stop();
      triggered = true;
    }

    public boolean hasBeenTriggered(){
      return triggered;
    }
}
```

Listing 4.4: TimeEventHandler Implementation

Both TimeEventHandler and TimeoutEventHandler are used to handle time events
attached to an activity. If the given time expression is a duration, then Timeout-
EventHandler will be used. If the given time expression is an exact time, Time-
EventHandler will be used. TimeEventHandler parses the given expression into
Java Date Object using the method `SimpleDateFormat.parse()`, while the Time-
outEventHandler will get a long integer parsed from the time expression.

4.3 Activites

Now we will discuss the activities in details. Activities are divided into tasks and subprocesses. As mentioned before, a task will be wrapped in an activity method. This enables each task to have their own scope of properties. The properties of subprocesses however, should be shared with all activities contained in it. Therefore wrapping subprocesses in a method is not enough. A subprocess will be wrapped in an inner class instead.

4.3.1 Tasks

Basically, the activity method generated from a task will look like what we can see in figure 4.9:

```
1  public void doTask(){
2    [declaration of properties]
3    [start assignments]
4    [special mapping depending on task-type]
5    [end assignments]
6  }
```
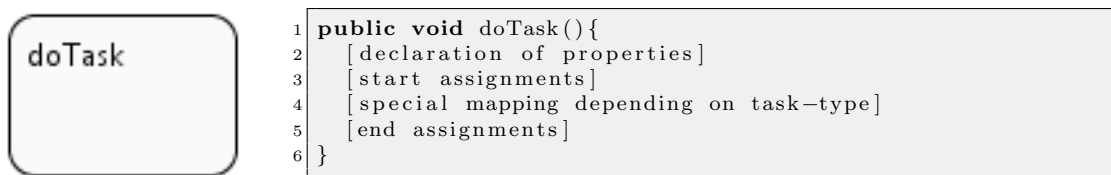
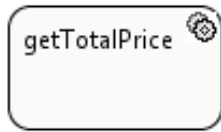Figure 4.9: Mapping Example: Task

The method will start by declaring Java variables, derived from the task's properties (if any exists). After the declaration, start assignments will take place followed by the task's actual mapping (if any, depending on the task type). Finally, the code will be closed by the end assignments. The BPMN allows us to specify properties and assignments (including when the assignment should take place) in the task's property sheet. Thus, unlike the workflow model in WADE it is possible to generate the operations needed to execute the activity completely from the model. Now let's take a closer look on how specific task types are being mapped.

Script-task
For Script-tasks, the script defined in the task's property will be directly added into the activity-method. This type of task is comparable to WADE's *Code Activity*, but just like the properties and assignments, with BPMN the script can be specified in the property sheet of the task. For the transformation to Agent Beans, the given script should be a valid Java expression.

Service-task
A service task is mapped into an invocation of an action defined by other Agent Bean. Figure 4.10 shows how a service task would be mapped. First, the action has to be found. Then the service will be invoked and the result will be assign to the outputs defined in the task's implementing service.

```
1  [...]
2  Action [operation] = retrieveAction ([interface].[operation
       ]);
3  Serializeable [] results = invokeAndWaitForResults ([
       operation], new Serializeable []{[inputs]}).getResults
       ();
4  [assign outputs from results]
5  [...]
```

Figure 4.10: Mapping Example: Service-task

Receive-task

A receive-task will be mapped into a Java code that reads the memory and wait until the specified message defined by the given MessageChannel is found in the memory.
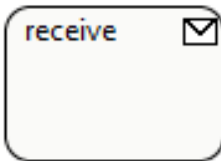
```
1  [...]
2  Action joinAction = retrieveAction (ICommunicationBean.
       ACTION_JOIN_GROUP);
3  Action leaveAction = retrieveAction (ICommunicationBean.
       ACTION_LEAVE_GROUP);
4  IGroupAddress groupAddress = CommunicationAddressFactory.
       createGroupAddress ([address]);
5  invoke(joinAction,new Serializable []{ groupAddress });
6  [payload] = null;
7  while([payload name]==null) {
8    Set<IFact> all = memory.readAll ();
9    for(IFact fact : all){
10     if(fact instanceof JiacMessage) {
11       JiacMessage jiacMessage = (JiacMessage)fact ;
12       if(jiacMessage.getPayload() instanceof Ping &&
             jiacMessage.getHeader(IJiacMessage.Header.
             SEND_TO).equals(groupAddress)) {
13         memory.remove(jiacMessage);
14         [payload name] = ([payload type]) jiacMessage.
               getPayload ();
15         break;
16       }
17     }
18   }
19   try{
20     Thread.sleep (100);
21   } catch(InterruptedException e) { }
22 }
23 invoke(leaveAction, new Serializable []{ groupAddress });
24 [...]
```

Figure 4.11: Mapping Example: receive-task

Send-task

A send-task will be mapped into an invocation of the ICommunicationBean's send action (see figure 4.12). The group address to which the message should be sent and the message itself will be derived from the given MessageChannel in the task's properties.

Call-task

A call-task will be mapped to an invocation of the called element. If the called element is an activity, then the activity method will be called. If the called element is a pool, then the call-task will be mapped into a service invocation.
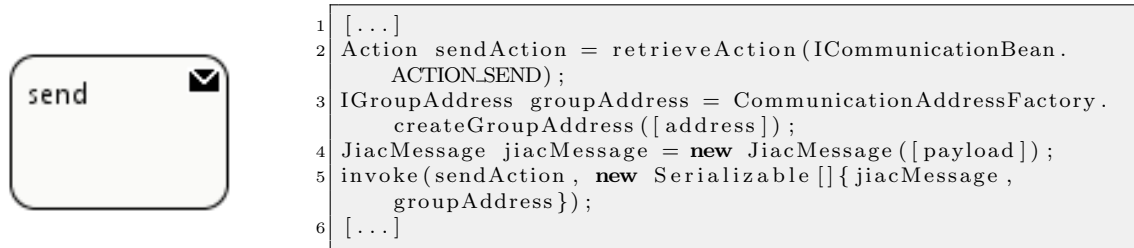
```
1  [...]
2  Action sendAction = retrieveAction(ICommunicationBean.
       ACTION_SEND);
3  IGroupAddress groupAddress = CommunicationAddressFactory.
       createGroupAddress([address]);
4  JiacMessage jiacMessage = new JiacMessage([payload]);
5  invoke(sendAction, new Serializable[]{jiacMessage,
       groupAddress});
6  [...]
```

Figure 4.12: Mapping Example: Send-task

User-task

The mapping of user tasks has not been completely developed yet. User-tasks require a user interface module to get inputs from the user. To make it simple we can put a TODO flag in the code, and the developer should implement the UI manually.

Manual-task

Manual-tasks will be executed manually by a person. Therefore, there is no need to develop a mapping to JIAC Agent Beans for manual-tasks.

Business-rule-task

The mapping of business-rule-tasks does not exist yet in this version.

4.3.2   Subprocess

A subprocess will be mapped into an inner class of the containing process or subprocess. This way its properties can be shared among all tasks contained in it. The inner subprocess will have the method `public void run()` containing the workflow (similar to the bean's workflow method). The following example shows the mapping of a subprocess:

```
1  //workflow−method
2  public void doProcess{
3  [...]
4  MySubProcess mySubProcess = new MySubProcess();
5  mysubProcess.run();
6  [...]
7  }
8  [...]
9  // subprocess as Innerclass
10 class MySubProcess {
11    [declaration of properties]
12    public void run(){
13      doA();
14      doB();
15    }
16    private void doA(){
17      [...]
18    }
19    private void doB(){
20      [...]
21    }
22 }
```
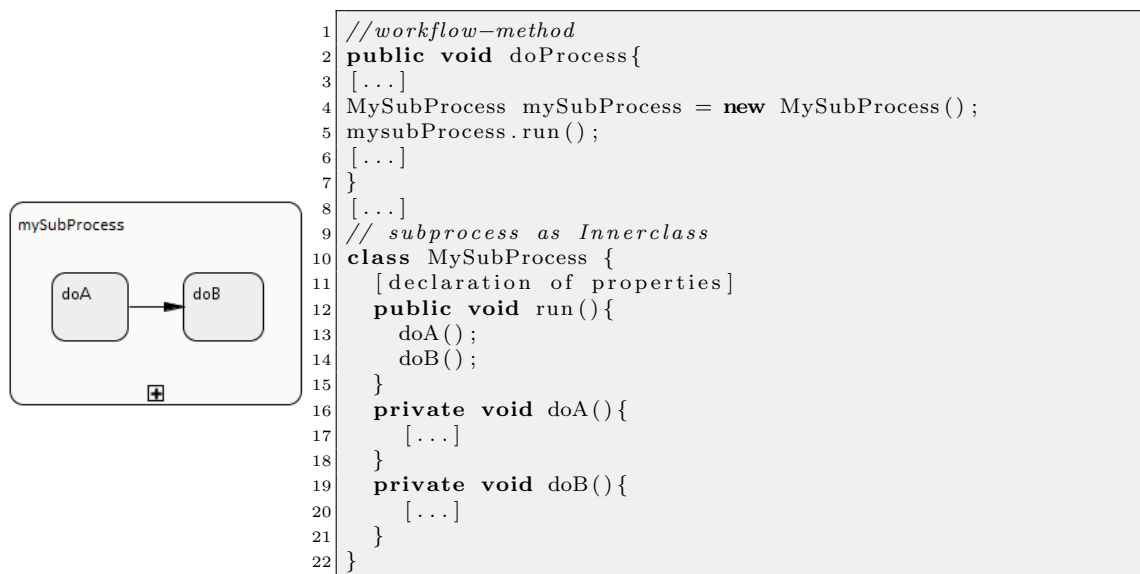
Figure 4.13: Mapping Example : Subprocess

In line 1-7 we can see that an instance of the subprocess class will be created in the Agent Bean's workflow method and afterwards the run method will be called.

4.3.3   Activity-Looping

An activity can have a looping property. This will result in the wrapping of the task-specific mapping in a while-block. There are two types of activity-looping in BPMN:

1. Standard-Loop

2. Multi-Instance-Loop

While the mapping of a standard-loop is trivial, the semantics of the multi-instance-loop is not very clear. Thus, a mapping of multi-instance-loop is not yet included in this version.

Standard-Loop
In a standard-loop, the task specific mapping will be wrapped in a while block. The other elements of the activity method (variable declarations and assignments) will not be wrapped. In figure 4.14 we can see an example of a script task with a standard loop.
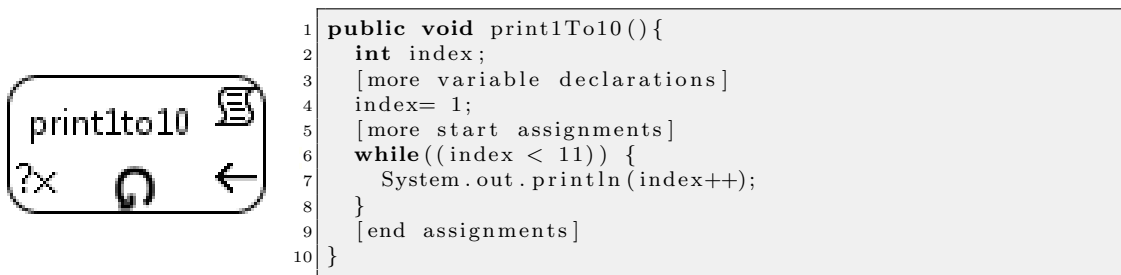
```
1  public void print1To10(){
2    int index;
3    [more variable declarations]
4    index= 1;
5    [more start assignments]
6    while((index < 11)) {
7      System.out.println(index++);
8    }
9    [end assignments]
10 }
```

Figure 4.14: Mapping Example : Activity-Looping (Standard-Loop)

4.4   Events

Finally, we will now discuss the mapping of BPMN events to JIAC Agent Beans. We will first discuss the intermediate events and then we will discuss start and end events. We need to group start and end events because the mapping will have influence on how the process will be started and they will also determine the signature (input and output parameters) of the workflow method.

4.4.1   Intermediate Events

Intermediate Events are something that happen during the process. Its mapping will be added into the workflow method. To keep the workflow method readable, we decided to wrap intermediate events in a method (similar to activities). The name will be derived from the event's name (if specified) or ID, added with an event-type specific postfix.

In this version only the mapping of timer and message intermediate events are included.

If the expression is a duration:

```
1  [...]
2  try {
3    Thread.sleep([time expression]);
4  } catch(InterruptedException e) {
5  }
6  [...]
```

If the expression is an exact time:

```
1  [...]
2  try {
3  Date then = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSSZ
       ").parse([time expression]);
4  long toSleep = then.getTime() - System.currentTimeMillis();
5  if(toSleep>0){
6    try {
7      Thread.sleep(toSleep);
8    } catch(InterruptedException e) {
9    }
10 }
11 } catch(ParseException e) {
12   System.out.println("ParseException: Time has to be in yyyy
       -MM-dd'T'HH:mm:ss.SSSZ form");
13   e.printStackTrace();
14 }
15 [...]
```

Figure 4.15: Mapping Example : Timer Intermediate Event

Timer

A timer intermediate event will turn the process to sleep until the given time expression. If the given time expression is a duration (option as duration in the property sheet is selected), then the expression will be handled as a long integer that defines how long (in milliseconds) the process should be paused.

If the given time is an exact time (e.g. Friday, November 4th 2011 10:00:00), then the expression should be parsed into a Java Date object and the process should be paused until the given date. However, it is not clear yet on how to handle incomplete date expressions (e.g. when we need to start a process every day at midnight). At the moment, the given date expression has to be in the form "yyyy-MM-dd'T'HH:mm:ss.SSSZ" (e.g. 2011-11-04T10:00:00.000+0100 for Friday, November 4th 2011 10:00:00). Figure 4.15 shows an example of the mapping for both variants.

Message

Message intermediate events will be mapped similarly to a receive-task. The process reads the memory and waits until the expected message is found in the memory.

4.4.2   Start and End Events

Timer

If a process starts with a timer event, then the workflow method will be called in the execute method. If the given time expression is a duration, the process will be started periodically. If the given time is an exact date, then the date will be parsed, and the process will be started when the date is passed. Figure 4.16 shows an example of the mapping for both variants.

If the expression is a duration:

```
1  public void execute(){
2    while(true){
3      [start process]
4      try {
5        Thread.sleep([time expression]);
6      } catch(InterruptedException e) {
7      }
8    }
9  }
```

If the expression is an exact time:

```
1  public void execute(){
2    try {
3        Date then = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:
            ss.SSSZ").parse("2011-11-04T10:00:00.000+0100");
4        long toSleep = then.getTime() - System.
            currentTimeMillis();
5        if(toSleep >=0){
6          try {
7            Thread.sleep(toSleep);
8          } catch(InterruptedException e) {
9          }
10         [start process]
11       }
12    } catch(ParseException e) {
13       System.out.println("ParseException: Time has to be in
            yyyy-MM-dd'T'HH:mm:ss.SSSZ form");
14       e.printStackTrace();
15    }
16 }
```

Figure 4.16: Mapping Example : Timer Start Event

Message

As mentioned before, message start and end events will influence the signature of the workflow method. The start event defines the parameters and the end event defines its return type. We can divide message start and end events according to the event's implementation. Currently, messages can be implemented by a service or a MessageChannel. If the implementation is a service, the communication describes a service call. Oherwise if the implementation is a MessageChannel, the communication describes a broadcast, which means a message will be sent to a communication group and all subscriber of the group will receive it.

Message with service implementation

If the implementation is a service, the workflow method will be exposed as an action (service). The process will be started by other Agent Bean as a service by invoking the exposed action. The signature of the workflow method will be derived from the signature of the implementing service. Multiple return types are wrapped in a serializable array.

In figure 4.17 we can see an example mapping of message start and end events with service implementation. The generated workflow method is exposed as an action with the @Expose code in line 5.
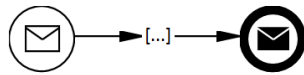
```
 1  public class [Bean−name] extends
        AbstractMethodExposingBean{
 2    public final static String ACTION_[workflow name] =
          "[bean fullname]#[workflow method]";
 3    [...]
 4
 5    @Expose(name = ACTION_[workflow name], scope =
          ActionScope.GLOBAL)
 6    public [service output] [workflow method]([service
          inputs]){
 7      [...]
 8      return [service output];
 9    }
10
11    [...]
12  }
```

Figure 4.17: Mapping Example : Message Start and End Events with Service Implementation

Message with MessageChannel implementation

If the implementation is a MessageChannel, an observer will be created and attached to the agent's memory in the `doStart()` method. The workflow method will get the MessageChannel's payload as a parameter. Further it will have no return type (void) and the result will be sent as a message to the specified MessageChannel of the end event. Figure 4.19 shows a mapping example of a message start event, and in figure 4.18 we can see a mapping example of how a message end event will be mapped.

```
 1  public class [Bean−name] extends AbstractMethodExposingBean{
 2    [...]
 3    public void [workflow method]([payload type] [payload name
          ]){
 4      [...]
 5      [payload type] [payload name]//payload variable
              declaration
 6      [payload name]= new StringOutput("ok");//payload
              assignment
 7      Action sendAction = retrieveAction(ICommunicationBean.
          ACTION_SEND);
 8      IGroupAddress groupAddress = CommunicationAddressFactory
          .createGroupAddress([channel]);
 9      JiacMessage jiacMessage = new JiacMessage([payload name
          ]);
10      invoke(sendAction, new Serializable[]{jiacMessage,
          groupAddress});
11    }
12    [...]
13  }
```

Figure 4.18: Mapping Example : Message End Event with MessageChannel Implementation

```
1   public class [Bean−name] extends AbstractMethodExposingBean{
2     [...]
3     public void doStart(){
4       Action joinAction = retrieveAction(ICommunicationBean.
            ACTION_JOIN_GROUP);
5       IGroupAddress groupAddress = CommunicationAddressFactory
            .createGroupAddress([channel]);
6       invoke(joinAction,new Serializable[]{groupAddress});
7       SpaceObserver<IFact> [event name or id]_observer = new
            SpaceObserver<IFact>(){
8         public void notify(SpaceEvent<? extends IFact> event)
              {
9           if(event instanceof WriteCallEvent<?>){
10            Object obj = ((WriteCallEvent)event).getObject();
11            if(obj instanceof IJiacMessage){
12              IJiacMessage message = (IJiacMessage)obj;
13              IFact payload = message.getPayload();
14              if(payload!=null && [payload name] instanceof [
                  payload type] && message.getHeader(
                  IJiacMessage.Header.SEND_TO).
                  equalsIgnoreCase([channel]){
15                memory.remove(message);
16                [workflow method]((([payload type])payload);
17              }
18            }
19          }
20        }
21      };
22      memory.attach(_nO3bwPwdEeCOWB3dJOsUA_observer);
23    }
24    public void [workflow method]([payload type] [payload name
          ]){
25      [...]
26    }
27    [...]
28 }
```

Figure 4.19: Mapping Example : Message Start Event with MessageChannel Implementation

4.5   Open Issues

In this chapter, we presented the details of the mapping from BPMN to JIAC Agent Beans. As mentioned in the previous sections, there are some elements that are not been mapped yet. In the future, we will need to furhter study the semantics of the unmapped elements (e.g. multi instance loop, rule events etc.) and develop their mappings.

Further there are other issues showing that BPMN is not sufficient to be applied in agent engineering. An example is the missing definition of data types. As mentioned in [16], BPMN does not support the definition of data types although they can be referenced in a process diagram. At the moment it is simply assumed that referenced data types exist.

Another problem is identified in the MessageChannel implementation. The MessageChannel only supports broadcasting, which means a message will be delivered to all subscribers of a communication group. In some scenarios, e.g the one we will present in chapter 6, we might need a message to be sent to a single agent.

5. Implementation of the Transformation

In this chapter we will present the details of the transformation's implementation. We will start with the slightly modified overall transformation structure and then continue with the details of each transformation stages. Finally we will present our solution for merging generated and manually added code, followed by some open issues regarding the current implementation.

5.1 Transformation Structure

As mentioned before in section 2.3, the transformation process in the VSDT is divided into five stages. We also mentioned that the default validation and structure mapping provided by the transformation framework are reusable. For the implementation of the transformation to Agent Beans, the framework's DefaultBpmnValidator and BPMN2StrucBPMNTransformation are being reused.
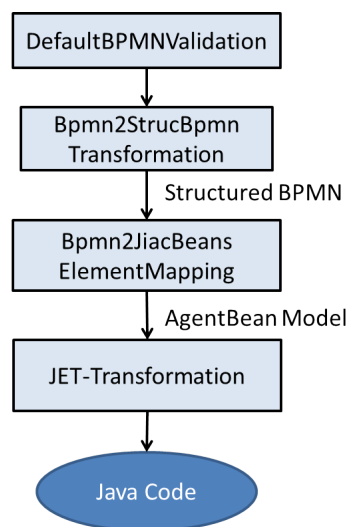
Figure 5.1: Transformation Stages

Figure 5.1 shows the structure of the transformation. Both normalization stage and structure mapping stage are implemented by the BPMN2StrucBPMN-Transformation. The clean-up stage is not included in this transformation. In the element mapping stage an intermediate Agent Bean model will be created. After the element mapping, the generated Agent Bean model will be passed into the JET-Transformation where it will be transformed into a Java file.

In the following sections we will take a closer look into the details of each transformation stage.

5.2 Validation

The DefaultBPMNValidation is currently used for the validation stage. However, it might be useful to implement a new validation that checks whether the expressions given in the model are conform with the Java syntax. At the moment, some expressions are handled in the ElementMapping stage e.g. changing types to have Java conformance. Some other expressions, e.g. content of a script task, are expected to be written in Java syntax.

5.3 Normalization and Structure Mapping

Similar to the validation stage, nothing new was implemented for the structure mapping stage. The default BPMN2StrucBPMN transformation of the VSDT responsible for performing these two stages is being reused.

In the normalization stage the process will be transformed into a semantically equivalent *normal form*. To achieve this, the transformation uses some rules e.g. the insert gateway rule that will insert a gateway for each activity with multiple incoming or outgoing sequence flows (see example in figure 5.2). The structure mapping stage is responsible for transforming the graph structured BPMN into an equivalent block structured representation. Both of the mentioned transformation stages are language independent, thus there is no need to implement a new transformation for these stages.
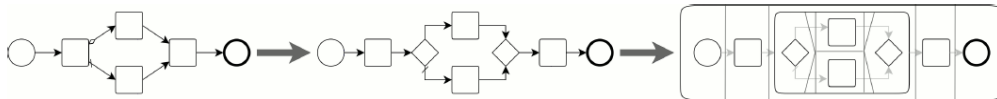

Figure 5.2: Simple Example of Normalization and Structure Mapping. [15]

5.4 Element Mapping

The element mapping is implemented with a visitor that will go through the elements of the process diagram with a top down traversion. In this stage, an Agent Bean Model representing a Java file holding an Agent Bean class will be generated for each pool visited in the diagram. Then the visitor will visit each element of the pool and populate the Agent Bean model with attributes and methods according to the mapping of each visited element.

For the implementation, the Agent Bean model was developed as an intermediate product. Let's take a closer look into the model.

5.4.1 Agent Bean Model

An Agent Bean model has a list of attributes, methods, actions and it may also have a list of subprocesses (because a subprocess is mapped into an inner class of the generated Agent Bean).

For the content of a method, a script model was also developed. A script is basically a Java code element, which can be a single CodeElement (a single line Java code), a sequence which contains a list of scripts or even a block construct such as the while loop or a try-catch block.
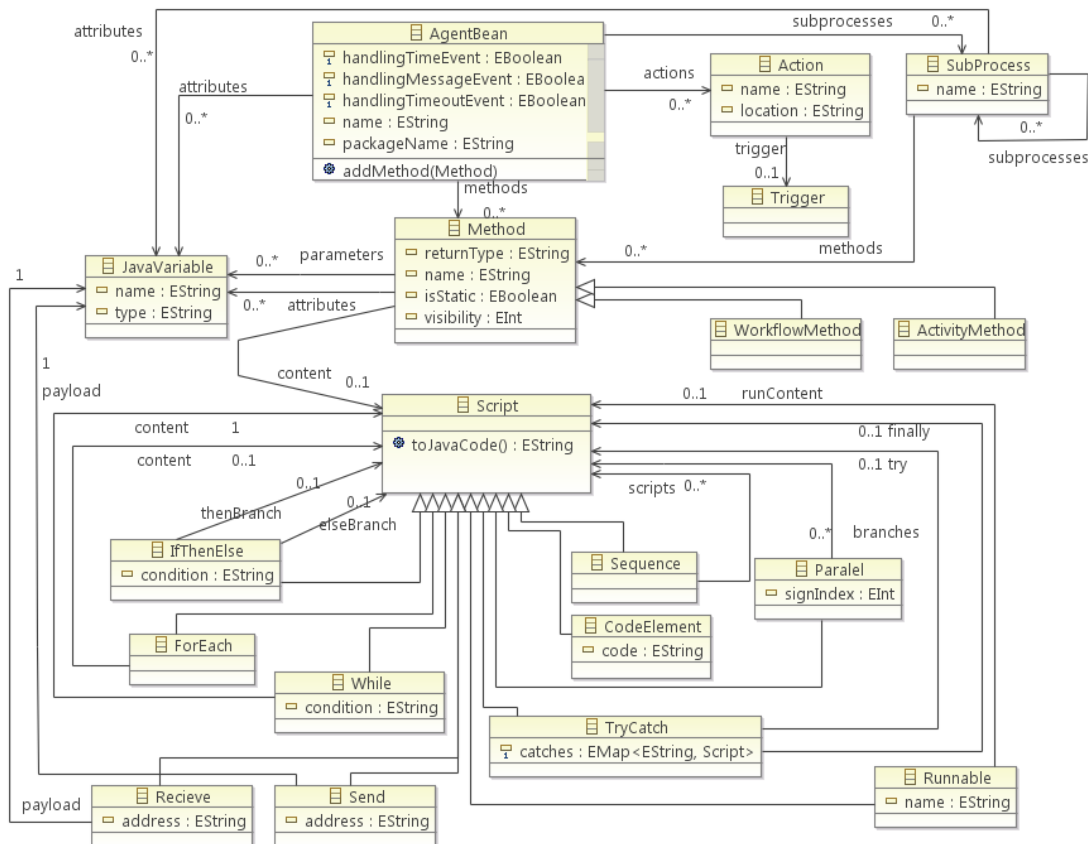

Figure 5.3: Agent Bean Model

Each script implements the method `public String toJavaCode()` which returns the java code representation of the script. In the following listing we can see the implementation of the method in the IfThenElse class:

```java
public String toJavaCode() {
  String code = "";
  code += "if("+condition+"){\n";
  if(thenBranch!=null){
    BufferedReader reader = new BufferedReader(new StringReader(thenBranch.
        toJavaCode()));
    try{
      String line = reader.readLine();
      while(line!=null){
        if(!line.equals("")) code += "\t"+line+"\n";
        line = reader.readLine();
      }
    }catch(IOException e){
      code += "\t//Error occured while reading if branch\n";
```

```
14          }
15        }
16      code+="}";
17      if(elseBranch!=null){
18        code+="else{\n";
19        BufferedReader reader = new BufferedReader(new StringReader(elseBranch.
            toJavaCode()));
20        try{
21          String line = reader.readLine();
22          while(line!=null){
23            if(!line.equals("")) code += "\t"+line+"\n";
24            line = reader.readLine();
25          }
26        }catch(IOException e){
27          code += "\t//Error occured while reading else branch\n";
28        }
29        code+="}";
30      }
31      return code;
32    }
```

Listing 5.1: toJavaCode() Implementation in the IfThenElse Class

You might notice that this method is also responsible for the text formatting because it will be used by the JET-Transformation and the result will then be written into a *.java file. For this purpose a tab is added in front of each line in the then and else branch, as you can see in line 7-11 and 21-25.

The Role of MDE in the Implementation

The benefits of Model Driven Engineering are also found during the implementation of the Agent Bean model. As we can see in figure 5.3, the model is created graphically using eclipse's Ecore Tools - Ecore Diagram. With the help of the EMF generator each element of the model can be easily generated into Java code including a factory class that can be used to instantiate an object of each generated class. This way we only have to implement the method toJavaCode() for each newly added script, everything else are generated automatically.

5.5   JET-Transformation

The JET-Transformation consists of five JET-Templates (see also figure 5.4):

1. agentbeantemplate.javajet

2. subprocesstemplate.javajet

3. timeouteventhandler.javajet

4. timeeventhandler.javajet

5. messageeventhandler.javajet

The agentbeantemplate is the main template of the implemented JET-Transformation. An instance of its Java template class is created by the JiacBeansResultSaver and the generate method will be invoked for each Agent Bean model generated in the element mapping stage.
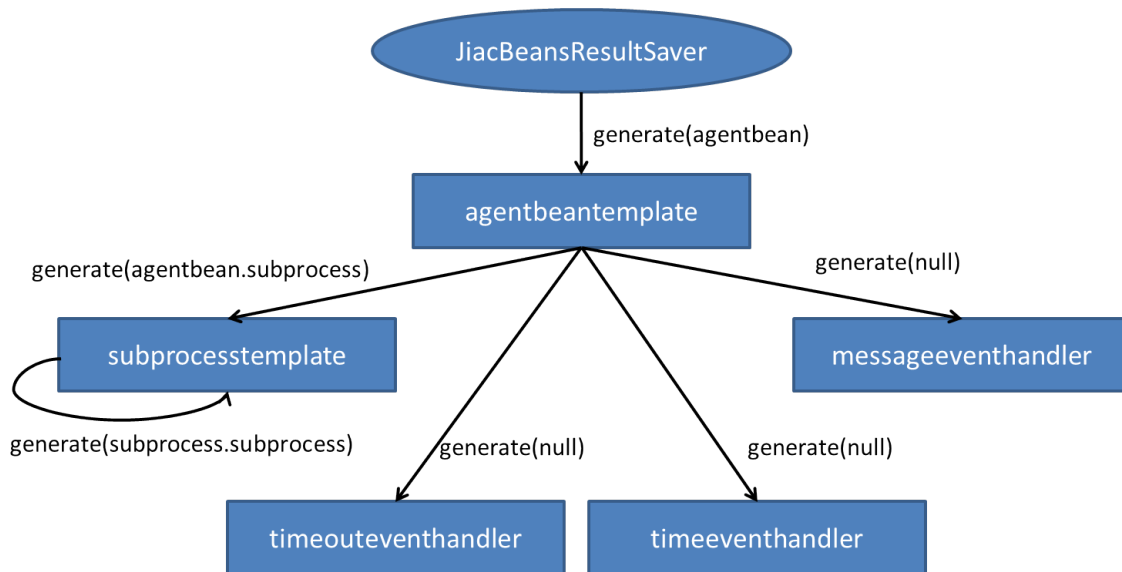
Figure 5.4: JET-Transformation Structure

The subprocesstemplate is invoked by the agentbeantemplate and recursively by the subprocesstemplate itself for each subprocess contained in their argument (an agentbean model or a subprocess model).

The three handler templates timeouteventhandler, timeeventhandler and messageeventhandler are static templates, which means the result of the generate method does not depend on the argument object. They are responsible for generating the event handler classes TimeoutEventHandler, TimeEventHandler and MessageEventHandler. They are invoked by the agentbeantemplate if the value of the flag handlingTimeoutEvent, handlingTimeEvent or handlingMessageEvent of the Agent Bean model is true.

5.6 Merging Generated and Manually Edited Code

The main challenge in generating Java code is how to handle code that has been manually edited. Fortunately, the EMF comes with a solution to this problem : **JMerge**[11]. With JMerge we can allow the regeneration of a model and protect manually added user code.

We can specify some rules in an XML-file that will be loaded by JMerge. The following listing 5.2 shows a very simple XML that can be used to differentiate generated elements from the the manually edited elements of a Java code.

```
1  <merge:dictionaryPattern
2      name="generatedMember"
3      select="Member/getComment"
4      match="\s*@\s*(gen)erated\s*\n"/>
5
6  <merge:pull
7      targetMarkup="^gen$"
8      sourceGet="Method/getBody"
9      targetPut="Method/setBody"/>
```

Listing 5.2: Example JMerge Rule

In this case, the dictionary pattern will match code members that have the @generated annotations somewhere in their comment. The string between the parentheses

in the match attribute defines the identifier which will be used in the pull element. The pull element in the example defines that the generator will replace the content of every method that has @generated in its comment.

The next thing to do is to add the comment with the @generated tags in our template for each generated elements (class, fields, methods). If a user edit the content of a method or the default value of a field, the @generated annotation should be deleted from the comment. Without the @generated annotation, the edited element will be let alone by JMerge and will not be lost after the regeneration of the code.

As seen in [19], there is a possibility to allow fine grained customization for methods and javadoc comments by using the pull elements shown in listing 5.3

```
1  <merge:pull
2     sourceMarkup="^gen$"
3     sourceGet="Member/getComment"
4     sourceTransfer="(\s*<!--\s*begin-user-doc.*?end-user-doc\s*-->\s*)\n"
5     targetMarkup="^gen$"
6     targetPut="Member/setComment"/>
7
8  <merge:pull
9     targetMarkup="^gen$"
10    sourceGet="Method/getBody"
11    sourceTransfer="(\s*//\s*begin-user-code.*?//\s*end-user-code\s*)\n"
12    targetPut="Method/setBody"/>
```

Listing 5.3: JMerge Example: fine grained customization for javadoc comments and methods

The first pull element will allow the user to put any comments between the `<!-- begin-user-doc -->` and the `<!-- end-user-doc -->` tags. The second one will allow the user to customize a method by adding code between the `// begin-user-code` and `// end-user-code` comments. Unfortunately, this does not seem to work if the above mentioned boundary tags and comments do not exist in the generated code. This means the position where the code may be customized should be identified in the template. For the javadoc there is no problem because the position does not matter, but for the content of a method the position cannot be predefined. Currently the javadoc comment shown in listing 5.4 will be added to each generated variable and method.

```
1  /**
2   *   <!-- begin-user-doc -->
3   *   <!-- end-user-doc -->
4   *   delete the generated tag after you edited this [field/method]
5   *   @generated
6   */
```

Listing 5.4: Javadoc Comment Used For JMerge

5.6.1   Using JMerge

In JET2, JMerge is included and can be used by adding the <java:merge/> tag somewhere in the template. However, because we are using the older version of JET we have to include JMerge in the `JiacBeansResultSaver` class. At the moment, JMerge can only be used in an eclipse plugin. Listing 5.5 shows us the code needed to use JMerge.

```
1  ASTFacadeHelper astFacadeHelper = new ASTFacadeHelper ( ) ;
2  JControlModel model = new JControlModel ( ) ;
3  model.setLeadingTabReplacement ( " \ t " ) ; // optional
4  // load the rules
5  model.initialize ( astFacadeHelper , getClass ( ) . getResource ( " mergerules.xml " ) . toString
        ( ) ) ;
6  JMerger jMerger = new JMerger ( model ) ;
7  // set the source ( new generated code )
8  jMerger.setSourceCompilationUnit ( jMerger.createCompilationUnitForContents ( content ) )
        ;
9  // set the target ( code from the existing file )
10 jMerger.setTargetCompilationUnit ( jMerger.createCompilationUnitForInputStream ( new
        FileInputStream ( f ) ) ) ;
11 // call the merge
12 jMerger.merge ( ) ;
13 // extract the merged text
14 String mergedContents = jMerger.getTargetCompilationUnit ( ) . getContents ( ) ;
```

Listing 5.5: Using JMerge

5.6.2 Problem with JMerge

If the generated code contains invalid imports, e.g. if the imported target is not in the classpath, an exception will be thrown. To avoid this, when calling the transformation to JIAC Agent Beans, the destination folder should be a source folder in a JIAC Project[1] so that all JIAC core elements needed by the Agent Bean are in the classpath. Otherwise, the code will not be merged and the existing file will be overwritten.

5.7 Open Issues

In this chapter we presented some details of the implementation. At the moment there are still some work that have to be done regarding the implementation of the transformation such as:

Complete the implementation of element mapping
The implementation of the element mapping is not yet complete. Some concepts such as the mapping of event based gateway still have to be implemented.

Mixing manually edited and generated code within a method
What we want is to let the user to edit only parts of the method's content by adding code within `// begin-user-code` and `// end-user-code`. For this purpose, either we should try to find a better mergerule or try to predefine the customizable position and use the rules shown in figure 5.3.

Find a better solution for JMerge exception
At the moment, the implementation will overwrite the existing code if JMerge fails to merge the code. To avoid losing manually edited code this should be changed. One solution could be by displaying a dialog where the user can choose whether to overwrite the code or to give another location for the new generated file.

---

[1]Please refer to the JIAC Manual[4] on how to create a JIAC Project

Identify modifiable and unmodifiable elements

In the current implementation the generated members are only marked with @generated. While some elements should be modifiable by the users, some others such as the workflow method should not. We can use @unmodifiable and @modifiable to differentiate whether an element may be edited.

Sort generated imports

Although it does not affect the validity of the generated code, the generated imports should be alphabetically sorted in order to improve the readability.

6. Example

In this chapter we will introduce an example scenario of a process and how the resulting code of the transformation looks like.

6.1 The Model

The example is based on a scenario taken from an actual topic: *electromobility*. The scenario includes a choreography between a MobileApplication, that will be used by a user to reserve a taxi and a number of ETaxi-applications installed in electronic taxis. Figure 6.1 shows the use case model of the **requestTaxi** process.
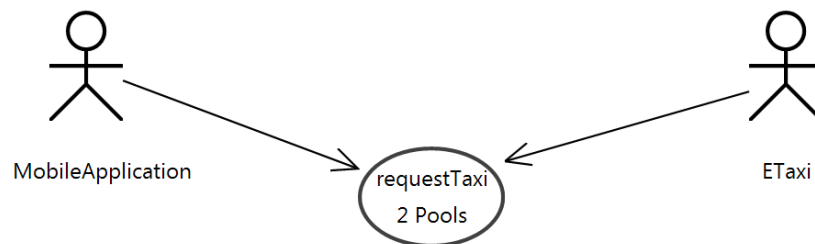

Figure 6.1: Use Case Model

In figure 6.1 we can see some data types included in the process which represent the information exchanged in the communication between both pools (the payload of the implementing MessageChannel). The transformation will assume that the classes exist and they will be imported by the generated Agent Bean. Further, because payloads will be wrapped in a JiacMessage, they have to implement the IFact interface.
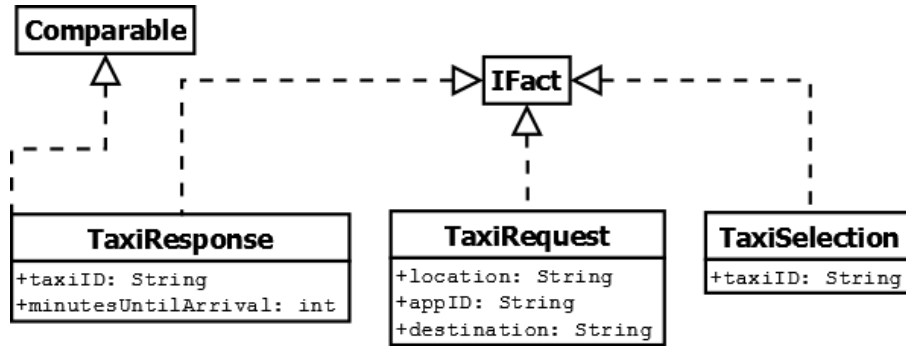
Figure 6.2: Payload data types included in the process diagram

6.1.1   The scenario

Before we start with the transformation, we will describe the scenario that lies behind the process model shown in figure 6.1.1 from the perspective of each pool.

Mobile Application
The process starts with a user requesting a taxi by invoking the service provided by the mobile application. The service will then make a request to all ETaxis by sending information about the user's location, application ID and the destination. Within 30 seconds after sending the request, the mobile application will collect responses from all available ETaxis. Each of these responses contains information about the taxi ID and the estimated time needed by this taxi to arrive at the user's current location. If multiple responses are received, the mobile application will record the best response according to the minimum estimated time. After 30 seconds the application will check whether a response was received. If yes, it will send a notification to all taxis informing which taxi is selected. Then the process will end and the selected taxi's ID will be sent to the user. If no respons is received, it will notify the user that no taxi is currently available.

ETaxi
From the ETaxi perspective, the process starts when a request from a mobile application is received. It will then evaluate the request and decide whether or not the request is interesting e.g according to the distance between the taxi's location and the user's location. If the request is interesting, the ETaxi will then send a response back to the requesting MobileApplication. After that it will wait for a notification from the MobileApplication and check the included taxiID. If the taxiID belongs to the ETaxi, the driver will be informed and the system will navigate to the user's location.

6.1.2   Problem With MessageChannel

As mentioned in the open issues of the mapping, with the MessageChannel implementation, a message will be broadcasted to all subscribers of a communication group. This is a problem for our scenario since the response from the ETaxis should be sent only to the requesting MobileApplication. In order to keep the scenario as it is, we decided to make a workaround for this problem and create a channel (communication group) where the application ID (appID) is included in the address. This way each mobile application will have its own channel.
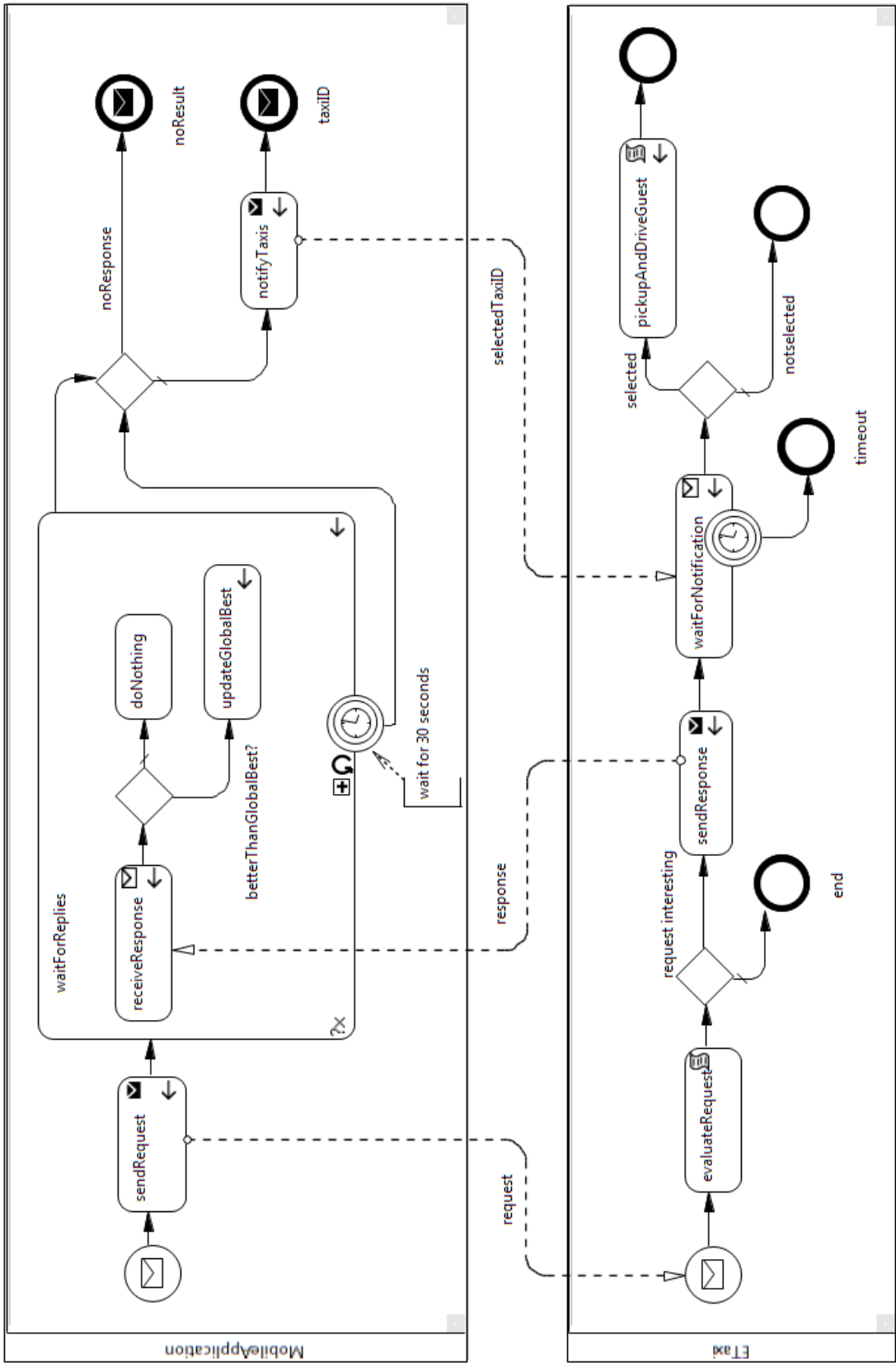
Figure 6.3: Business Process Diagram - requestTaxi

At the moment, the implemented element mapping will map the given channel into a String literal (the expression will be wrapped with two quotation marks `"[expression]"`). Obviously we need to avoid the wrapping of the appID variable as a String literal. Therefore we constructed the channel expression as `TaxiResponseTo"+appID+"`. This way the mapped expression will be `"TaxiResponseTo"+appID+""`. Perhaps it would be better if we change the implementation in the future and skip the wrapping of the given expression. Instead we can simply assume that the given expression is a valid Java String literal or a String variable e.g `"TaxiRequest"` or `requestChannel`(requestChannel must be a String variable contained in the Agent Bean).

6.2 The Generated Agent Beans

In this section we will present the Agent Beans generated from the above model. To save space and improve the readability, the generated code will be reduced as follows:

- Imports are deleted

- Javadoc elements used for JMerge are deleted

- The inner class TimeoutEventHandler which we already presented in section 4.2.4 is deleted

- Some empty lines are deleted

Beside the mentioned reduction, no other changes were made to the generated code. All codes and comments are generated directly from the model.

1. MobileApplication_requestTaxi

```
1  package mobileapplication;
2  [imports]
3  public class MobileApplication_requestTaxi extends AbstractMethodExposingBean{
4    public final static String ACTION_REQUESTTAXI = "mobileapplication.
         MobileApplication_requestTaxi#requestTaxi";
5
6    TaxiResponse globalBest;
7    boolean noResponse;
8    String appID;
9    String location;
10   String destination;
11
12   @Expose(name = ACTION_REQUESTTAXI, scope = ActionScope.GLOBAL)
13   public String requestTaxi(String currentLocation, String currentDestination) {
14     location= currentLocation;
15     destination= currentDestination;
16     sendRequest();
17     Thread waitForReplies = new Thread(new Runnable(){
18       public void run(){
19         WaitForRepliesSubProcess waitForReplies = new WaitForRepliesSubProcess();
20         waitForReplies.run();
21       }
22     });
23     TimeoutEventHandler wait30Seconds_TimeoutHandler = new TimeoutEventHandler
           (30000,waitForReplies);
24     waitForReplies.start();
25     wait30Seconds_TimeoutHandler.start();
26     try {
```

```
27        waitForReplies.join();
28        wait30Seconds_TimeoutHandler.stop();
29      } catch(InterruptedException e) {
30      }
31
32      if(noResponse){
33        String taxiID;
34        taxiID= "none available";
35        return taxiID;
36      } else {
37        notifyTaxis();
38        String taxiID;
39        taxiID= globalBest.getTaxiID();
40        return taxiID;
41      }
42    }
43
44    private void sendRequest() {
45      TaxiRequest request;
46      request= new TaxiRequest(location,appID,destination);
47      Action sendAction = retrieveAction(ICommunicationBean.ACTION_SEND);
48      IGroupAddress groupAddress = CommunicationAddressFactory.createGroupAddress("
             TaxiRequest");
49      JiacMessage jiacMessage = new JiacMessage(request);
50      invoke(sendAction, new Serializable[]{jiacMessage, groupAddress});
51    }
52
53    private void notifyTaxis() {
54      SelectedTaxi selectedTaxi;
55      selectedTaxi= new SelectedTaxi(globalBest.getTaxiID());
56      Action sendAction = retrieveAction(ICommunicationBean.ACTION_SEND);
57      IGroupAddress groupAddress = CommunicationAddressFactory.createGroupAddress("
             notification");
58      JiacMessage jiacMessage = new JiacMessage(selectedTaxi);
59      invoke(sendAction, new Serializable[]{jiacMessage, groupAddress});
60    }
61
62    class WaitForRepliesSubProcess{
63
64      TaxiResponse currentResponse;
65
66      public void run() {
67        noResponse= true;
68        while(true) {
69          receiveResponse();
70          if(currentResponse.compareTo(globalBest)>0){
71            updateGlobalBest();
72          } else {
73            doNothing();
74          }
75        }
76      }
77
78      private void receiveResponse() {
79        Action joinAction = retrieveAction(ICommunicationBean.ACTION_JOIN_GROUP);
80        Action leaveAction = retrieveAction(ICommunicationBean.ACTION_LEAVE_GROUP);
81        IGroupAddress groupAddress = CommunicationAddressFactory.createGroupAddress("
               TaxiResponseTo"+appID+"");
82        invoke(joinAction,new Serializable[]{groupAddress});
83        TaxiResponse response = null;
84        while(response==null) {
85          Set<IFact> all = memory.readAll();
86          for(IFact fact : all){
87            if(fact instanceof JiacMessage) {
88              JiacMessage jiacMessage = (JiacMessage)fact;
89              if(jiacMessage.getPayload() instanceof TaxiResponse && jiacMessage.
                     getHeader(IJiacMessage.Header.SEND_TO).equals("TaxiResponseTo"+
                     appID+"")) {
90                memory.remove(jiacMessage);
91                response = (TaxiResponse) jiacMessage.getPayload();
92                break;
93              }
94            }
95          }
```

```
96          try{
97            Thread.sleep(100);
98          } catch(InterruptedException e) { }
99        }
100       invoke(leaveAction, new Serializable[]{groupAddress});
101       noResponse= false;
102       currentResponse= response;
103     }
104
105     private void doNothing() {
106     }
107
108     private void updateGlobalBest() {
109       globalBest= currentResponse;
110     }
111   }
112 }
```

Listing 6.1: Generated Agent Bean - MobileApplication_requestTaxi

In the MobileApplication_requestTaxi Bean, almost every logic is generated from the model. The workflow method requestTaxi is exposed as an action. All methods are filled with needed operations e.g. the decision whether the currentResponse is better than the globalBest is made by using the compareTo method of the TaxiResponse class. There is only one small yet fatal aspect missing in the generated code. You might notice that while other variables are initialized somewhere in the assignment of a method, the initialization of the appID variable can't be found in the generated code. To ensure that each instance of the MobileApplication_requestTaxi has a unique ID, the initialization cannot be included in the model. We have to e.g. add a constructor and implement the initialization of the appID in it to make the code works completely. This problem also exists in the generated ETaxi_requestTaxi code.

2. ETaxi_requestTaxi

```
1  package etaxi;
2  [imports]
3  public class ETaxi_requestTaxi extends AbstractMethodExposingBean{
4    public final static String ACTION_REQUESTTAXI = "etaxi.ETaxi_requestTaxi#
         requestTaxi";
5
6    boolean requestInteresting;
7    String taxiID;
8    String globalLocation;
9    int estimatedTime;
10   String appID;
11   TaxiRequest currentRequest;
12   SelectedTaxi selection;
13   boolean available;
14
15   @Expose(name = ACTION_REQUESTTAXI, scope = ActionScope.GLOBAL)
16   public void requestTaxi(TaxiRequest request) {
17     currentRequest= request;
18     evaluateRequest();
19     if(requestInteresting){
20       sendResponse();
21       Thread waitForNotification = new Thread(new Runnable(){
22         public void run(){
23           waitForNotification();
24         }
25       });
26       TimeoutEventHandler _x5RCUAGWEeGC8PuSIWxlmQ_TimeoutHandler = new
            TimeoutEventHandler(60000, waitForNotification);
27       waitForNotification.start();
28       _x5RCUAGWEeGC8PuSIWxlmQ_TimeoutHandler.start();
29       try {
30         waitForNotification.join();
```

```
31        _x5RCUAGWEeGC8PuSIWxlmQ_TimeoutHandler.stop();
32      } catch(InterruptedException e) {
33      }
34      if(!_x5RCUAGWEeGC8PuSIWxlmQ_TimeoutHandler.hasBeenTriggered()){
35        if(selection.getTaxiID().equals(taxiID)){
36          pickupAndDriveGuest();
37        } else {
38        }
39      }
40    } else {
41    }
42  }
43
44  public void doStart() {
45    Action joinAction = retrieveAction(ICommunicationBean.ACTION_JOIN_GROUP);
46    IGroupAddress groupAddress = CommunicationAddressFactory.createGroupAddress("
          TaxiRequest");
47    invoke(joinAction,new Serializable[]{groupAddress});
48    SpaceObserver<IFact> _B8QyYAF3EeGC8PuSIWxlmQ_observer = new SpaceObserver<IFact
          >(){
49      public void notify(SpaceEvent<? extends IFact> event) {
50        if(event instanceof WriteCallEvent){
51          Object obj  = ((WriteCallEvent) event).getObject();
52          if (obj instanceof IJiacMessage){
53            IJiacMessage message = (IJiacMessage)obj;
54            IFact payload = message.getPayload();
55            if(payload!=null && payload instanceof TaxiRequest&& message.getHeader(
                  IJiacMessage.Header.SEND_TO).equalsIgnoreCase("TaxiRequest")){
56              memory.remove(message);
57              requestTaxi((TaxiRequest)payload);
58            }
59          }
60        }
61      }
62    };
63    memory.attach(_B8QyYAF3EeGC8PuSIWxlmQ_observer);
64  }
65
66  private void evaluateRequest() {
67    //script activity
68    //TODO implement code
69    requestInteresting = true; //every request is interesting
70  }
71
72  private void sendResponse() {
73    TaxiResponse response;
74    response= new TaxiResponse(taxiID, estimatedTime);
75    Action sendAction = retrieveAction(ICommunicationBean.ACTION_SEND);
76    IGroupAddress groupAddress = CommunicationAddressFactory.createGroupAddress("
          TaxiResponseTo"+appID+"");
77    JiacMessage jiacMessage = new JiacMessage(response);
78    invoke(sendAction, new Serializable[]{jiacMessage, groupAddress});
79  }
80
81  private void waitForNotification() {
82    Action joinAction = retrieveAction(ICommunicationBean.ACTION_JOIN_GROUP);
83    Action leaveAction = retrieveAction(ICommunicationBean.ACTION_LEAVE_GROUP);
84    IGroupAddress groupAddress = CommunicationAddressFactory.createGroupAddress("
          notification");
85    invoke(joinAction,new Serializable[]{groupAddress});
86    SelectedTaxi selectedTaxi = null;
87    while(selectedTaxi==null) {
88      Set<IFact> all = memory.readAll();
89      for(IFact fact : all){
90        if(fact instanceof JiacMessage) {
91          JiacMessage jiacMessage = (JiacMessage)fact;
92          if(jiacMessage.getPayload() instanceof SelectedTaxi && jiacMessage.
                getHeader(IJiacMessage.Header.SEND_TO).equals("notification")) {
93            memory.remove(jiacMessage);
94            selectedTaxi = (SelectedTaxi) jiacMessage.getPayload();
95            break;
96          }
97        }
98      }
```

```
 99          try{
100              Thread.sleep(100);
101          } catch(InterruptedException e) { }
102      }
103      invoke(leaveAction, new Serializable[]{groupAddress});
104      selection= selectedTaxi;
105    }
106
107    private void pickupAndDriveGuest() {
108      available= false;
109      // TODO add code to navigate to guest's location
110      available= true;
111    }
112
113 }
```

Listing 6.2: Generated Agent Bean - ETaxi_requestTaxi

Similar to the MobileApplication_requestTaxi Bean, the initialization of the taxiID is also missing in the generated code. Moreover, unlike the MobileApplication_request-Taxi Agent Bean, we did not include all the logic in the model (e.g. for evaluating whether a request is interesting). By this we showed that the quantity of the generated code might vary depending on how we design the model.

In this chapter we have presented an example of the transformation in which some problems regarding the current mapping and transformation are shown. We have also shown how agents can be designed and created easily by modeling a business process.

7. Conclusion

In this work, we combined two MDE approaches in the domain of multi-agent-systems and brought the benefits of the expressive model BPMN and the developer friendly Java programming language together. As a result, a plugin to VSDT that adds a transformation from BPMN to JIAC Agent Beans was developed.

First, we presented some information about the target framework JIAC, the used model BPMN, the existing model driven tool VSDT and about the code generating framework JET. Further we also presented the result of our research about the related state of the art approaches and existing transformations of BPMN to BPEL and JADL.

Inspired by the BPMN to JADL transformation and WADE, we then developed the mapping from BPMN to JIAC Agent Beans as a combination from both approaches. Similar to the presentation of the mapping from BPMN to BPEL that we found in the BPMN specification, we used examples that show an excerpt of the process and the associated code mapped from it to present the mapping from BPMN to Agent Beans.

The transformation was developed with the benefit of the extensible transformation framework of the VSDT, where we could reuse the built in transformations for the validation, normalization and structure mapping stages. We have shown that VSDT's transformation framework can be extended easily by implementing a transformation for the element mapping stage. To simplify the transformation, an Agent Bean model was also developed as an intermediate product. During the development process of the Agent Bean model we gained the benefit of MDE provided by eclipse's EMF that enabled us to generate most of the Java code for the Agent Bean model automatically. To protect the manually added code while allowing a regeneration of the model, we used the help of another EMF solution JMerge.

Finally we presented an example and showed that our solution has made it possible for agents to be created by designing a BPMN process. This should make it easier for people from the business world to design a software agent or multi agent systems. Further the models created to design the agents can also be used in the communication with non experts e.g when presenting the designed multi agent systems to the stakeholders.

7.1   Future Work

As mentioned in the previous chapters, there are still some problems and open issues regarding the presented solution. Derived from these problems we have identified some work packages that need to be done in the future.

7.1.1   Complete the Mapping

The mapping discussed in chapter 4 is not complete. The next work to be done is to develop a concept for the missing mapping. According to our experience in developing the mapping, it is very helpful to experiment with a self-implemented Agent Bean to check whether the code we concepted is doing what we expect it to do.

After all mapping is developed the element mapping has to be implemented in the transformation.

7.1.2   Solve Open Issues of the Implementation

As mentioned in section 5.7, there are some open issues in the implementation that need to be solved. Most of these issues can be solved easily and should be done in near future.

The most urgent work that needs to be done is to complete the implementation of the existing mapping concept such as the mapping of event_based gateway. The next urgent work is to provide a better merging solutions to support fine grained customization of the methods. We should also differentiate modifiable and unmodifiable elements of the generated code to prevent the modification of key elements such as the workflow method by the user.

Further we need to implement the validation stage to check the validity of the expressions regarding the Java syntax and sort the generated imports to improve the readability of the generated code.

7.1.3   Support Single Agent Communication

To give a better support single agent communication and to avoid workarounds, we might need to add another implementation of messages. One solution could be made by adding a *reply-to* address in the JiacMessage. This way even if the communication must start with a broadcast request, the response can be sent directly to the requesting agent and so is each following communication.

7.1.4   JET2 Migration

For the future we might have to study the possibility about migrating the JET-Transformation to JET2. But because we have successfully implemented the transformation using the older version of JET, this work has a very low priority and we should really consider whether the migration to JET2 would be beneficial.

7.1.5   Round-Tripping

Importing any manually implemented Agent Bean would not be applicable, because Java is a very expressive language and there are many ways to implement a concept. However, it would be very nice if the model and the generated Java code can be kept in sync like the WADE-workflow. To achieve this, we might consider to learn something from WADE's approach and develop a special Agent Bean with a well-defined structure that matches the BPMN.

Bibliography

[1] Fabio Bellifemine, Giovanni Caire, Agostino Poggi, and Giovanni Rimassa. Jade: A software framework for developing multi-agent applications. lessons learned. *Inf. Softw. Technol.*, 50:10–21, January 2008.

[2] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE - a FIPA-compliant agent framework. In *Proceedings of the Practical Applications of Intelligent Agents*, 1999.

[3] Giovanni Caire, Danilo Gotta, and Massimo Banzi. Wade: a software platform to develop mission critical applications exploiting agents and workflows. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: industrial track*, AAMAS '08, pages 29–36, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[4] CC ACT, DAI-Labor, TU-Berlin. *JIAC - Java Intelligent Agent Componentware*, 06 2010. Version 5.1.0 Manual.

[5] Object Management Group. Bpmn specification v2.0. `http://www.omg.org/spec/BPMN/2.0/`.

[6] Benjamin Hirsch, Thomas Konnerth, and Axel Heßler. Merging agents and services - the jiac agent platform. In Amal El Fallah Seghrouchni, Jürgen Dix, Mehdi Dastani, and Rafael H. Bordini, editors, *Multi-Agent Programming:*, pages 159–185. Springer US, 2009. 10.1007/978-0-387-89299-3_5.

[7] Eclipse Modelling Project Homepage. `http://www.eclipse.org/modeling/`.

[8] JIAC Homepage. `http://jiac.de/?id=35`.

[9] VSDT Homepage. `http://jiac.de/?id=30`.

[10] WADE Homepage. `http://jade.tilab.com/wade/index.html`.

[11] What is JMerge. `http://wiki.eclipse.org/JET_FAQ_What_is_JMerge%3F`.

[12] JET. `http://www.eclipse.org/modeling/m2t/?project=jet#jet`.

[13] Thomas Konnerth, Benjamin Hirsch, and Sahin Albayrak. JADL — an agent description language for smart agents. In Mateo Baldoni and Ulle Endriss, editors, *Declarative Agent Languages and Technologies IV*, volume 4327 of *LNCS*, pages 141–155. Springer Verlag, 2006.

[14] Tobias Küster. Development of a visual service design tool providing a mapping from BPMN to JIAC. Master's thesis, Technical University of Berlin, Faculty of Electrical Engineering and Computer Science, 2007.

[15] Tobias Küster and Axel Heßler. Towards transformations from BPMN to heterogeneous systems. In Massima Mecella and Jian Yang, editors, *BPM2008 Workshop Proceedings*, 2008.

[16] Tobias Küster, Marco Lützenberger, Axel Heßler, and Benjamin Hirsch. Integrating process modelling into multi-agent system engineering. In Michael Huhns, Ryszard Kowalczyk, Zakaria Maamar, Rainer Unland, and Bao Vo, editors, *Proceedings of the 5th Workshop of Service-Oriented Computing: Agents, Semantics, and Engineering (SOCASE) 2010*, 2010.

[17] Chun Ouyang, Marlon Dumas, Arthur Ter Hofstede, and Wil M. P. Van Der Aalst. From BPMN Process Models to BPEL Web Services. pages 285–292, September 2006.

[18] Chun Ouyang, Wil M. P. van der Aalst, Marlon Dumas, and Arthur H.M. ter Hofstede. Translating BPMN to BPEL. Technical Report BPM-06-02, BPM Center, 2006.

[19] Adrian Powell. Customize generated models and editors with eclipse's jmerge. `http://www.ibm.com/developerworks/library/os-ecemf3/`. *Model with the Eclipse Modeling Framework, Part 3*, 2004.

[20] Eclipse Help Page Tag Library Reference. `http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.jet.doc/references/taglibs/index.xhtml`.

[21] Giovanni Caire (Telecom Italia S.p.A). *WADE User Guide*. Copyright ©2010, Telecom Italia, July 2010.

[22] Matthias Weidlich, Gero Decker, Alexander Grosskopf, and Mathias Weske. BPEL to BPMN: The Myth of a Straight-Forward Mapping. In *Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part I on On the Move to Meaningful Internet Systems:*, OTM '08, pages 265–282. Springer-Verlag, 2008.