DAI-Labor
TU Berlin

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.

Ort, Datum	Petrus Setiawan Tan

Abstract

Zusammenfassung

Acknowledgement

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Introduction

In this chapter, we will start by introducing the motivation and the goals of this work.

1.1 Motivation

A common problem in the software engineering is the communication gaps between the business people as a client and the IT world. Communicating through models, normally consists of graphical skectches, especially using those which are commonly used in the client's domain such as the BPMN, clearly is a solution in bridging these gaps.

The main motivation of this work is to present a solution that will simplify the software development process and help bringing the concepts of software agents and multi agent systems, a topic which has been researched for decades, to gain more acceptance in the industry by using the beneits of Model Driven Engineering(MDE).

1.1.1 Model Driven Engineering (MDE)

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 define the software. By modelling the software, the developer creates documents that provides an abstract view of the software system, independently from the plattform or a specific programming language, making it understandable for non experts i.e. the stakeholders as well as applicable in different plattforms.

A significant number of the so called CASE (Computer Aided Software Engineering) tools have 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 allows us to transform the model into text or even executeable Programs, thus increasing efficiency in the software development process.

We can say that the real benefits 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 by Tobias Küster in scope of his Diploma Thesis, which provides the transformation of BPMN (Business Process Modelling Notation) to BPEL(*Business Process Execution Language*) and JIAC (Java-based Intelligent Agent Componentware) framework. This tool allows agents to be designed using a business process diagram, a model which has already been manifested in the industry, thus most people from the industry will be able to design software agents. In the scope of this work, a plugin to VSDT will be developed to enrich it's transformation feature with a code generator that will transform BPMN models into executeable Java Code, or JIAC AgentBeans to be more specific.

1.2 Goals

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 Java Code or JIAC AgentBeans to be more specific. 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 has to be made, such that conflicts should not occur when the transformation is called to a code that has been edited manually.

2. Background

In this chapter, we will discuss the backgrounds of the transformation that should be developed. We will start with JIAC (the target of the transformation), BPMN (the model that is being used), followed by VSDT (the existing modeling and transformation framework to JIAC that should be extended), and JET (the technology that will help us to implement the transformation).

2.1 JIAC

JIAC (Java-based Intelligent Agent Componentware) is a Java-based agent architecture and framework that was developed to simplify the development of software agents[9]. The framework supports the entire software development process of a software agent system by providing features such as FIPA compliant communication, Believe-Desire-Intention (BDI) reasoning, strong migration, web-service connectivity and many others. It also provides high security (Common Criteria EAL3, certified by the Federal Office for Information Security of Germany, BSI) and advanced accounting mechanisms, making it suitable for the use in industrial and commercial applications.

JADL
With it's core component JADL(*Jiac Agent Description Language*)[14], Jiac also provide an agent programming language. With JADL, the agents *plan elements, rules, ontologies* and *services* can be described. JADL is based on three predicate logic which allows the values *true, false and unknown*, which makes it suitable for open world problems in unknown environments.

In Figure 2.1, we can see the typical structure of a JIAC Application, which consists of AgentNodes, Agents, AgentBeans. An *AgentNode* is a Java VM where the runtime infrastructure for agents, such as discovery services, white and yellow pages services, as well as communication infrastructure, are provided. A JIAC application might consists of multiple AgentNodes (distributed application). With the so-called

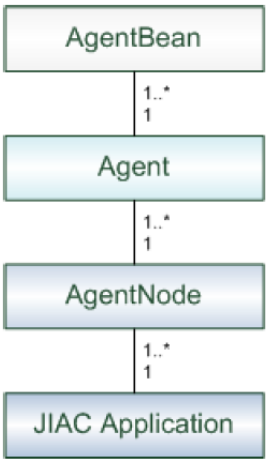


Figure 2.1: Jiac Basic concepts and their structural relationships [4]

AgentNodeBeans, one can extend the AgentNode with additional components.

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

2.1.1 AgentBeans

Beside using JADL, another mean to implement the functionality of an agent by implementing an agent bean 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 Agents 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 mehtods are implemented by the class `AbstractLifeCycle`, a super class of `AbstractAgentBean` which also provides the methods `doInit()`, `doStart()`, `doStop()`, and `doCleanup()` where we can hook up code that should be executed when the Lifecycle state changes, for example looking up needed actions, attaching SpaceObeserver to the agent's memory, etc.

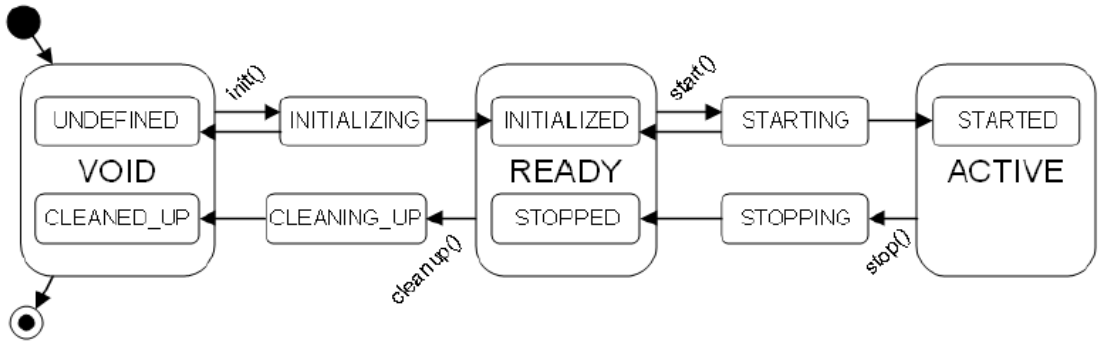


Figure 2.2: Agent-Lifecycle [4]

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 intervall 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. invoking 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, reciever})//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]Informations 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.Serializeable). The memory provides the 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 that are fired:

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

2.2 BPMN(Business Process Modelling Notation)

BPMN [5] is a standard Notation for modelling business processes, initially published by the BPMI which is later adopted by the OMG(Object Management Group). A business process diagramm (as seen in Figure 2.3) can be compared to UML's activity 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

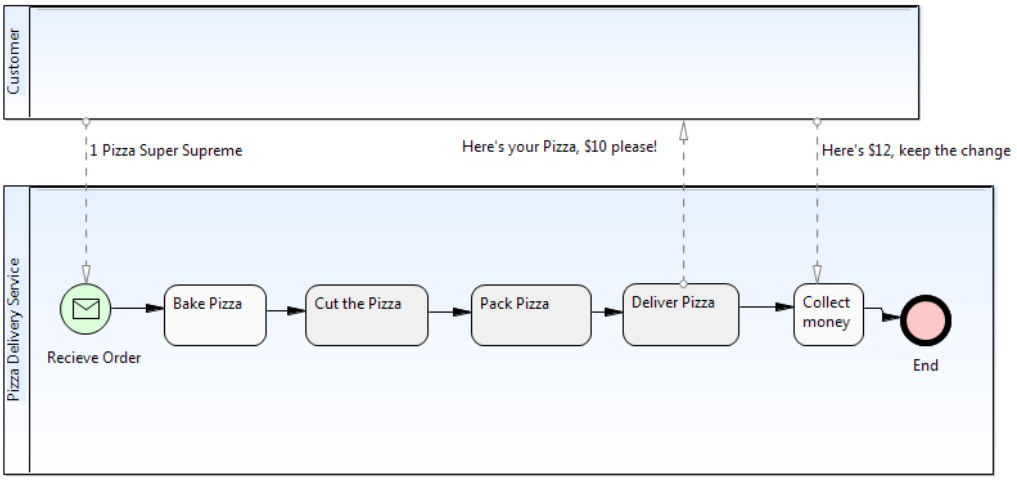


Figure 2.3: A simple Business Process Diagramm

implementation. Besides providing an easy to read graphical notations, each BPMN elements are also equipped with additional attributes (so called properties), which are hidden from the diagram and provides the informations needed for automated code generation. 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

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 marks the type of an event's trigger.
3. Properties - Each element has additional detailed informations that are hidden from the diagram, but contain informations needed for the transformation.

2.2.2 BPMN Elements

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

- Flow Objects
- Data
- Connecting Objects
- Swimlanes
- Artifacts

Figure 2.4: BPMN Event types. From left to right: Start Event, Intermediate Event, End Event

Flow Objects includes 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).
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 figure (see Figure 2.5) are drawn with different icon 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).
Gateways are used to define all kinds of splitting and merging behavior. It's semantics depends on the dimension of it's incoming and outgoing Sequence Flows.

Data is representated by the 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 *ConnectingObjects*, FlowingObjects can be connected to each other or with other informations. There are 3 different kinds of Connecting Objects:

- *Sequence Flows* - represent Flow Control, used for connecting FlowObjects within a Pool in the order of execution.
- *Message Flows* - represent Messages being exchanged exclusively between Pools.
- *Associations* - mainly used for documentation, in example 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 partition a Pool, in example to model different Departments of an Institution. Figure 3 shows an empty Pool with 2 Lanes.

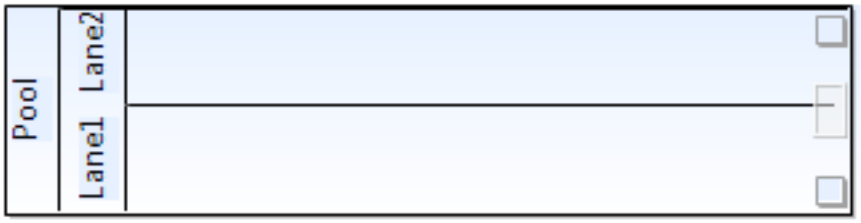


Figure 2.6: Empty Pool with 2 Lanes

Artifacts are additional Information about the Process. It is mainly used for documentation purpose. The Current set of Artifacts includes:

- Text Annotation
- Group

To provide a rough overview on how 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 are able to communicate with other agents (another pool) through messages (represented with the BPMN MessageFlows). Agents can react to Events. A detailed mapping needed for implementing the transformation will be discussed in chapter 4

2.3 VSDT(Visual Service Design Tool)

The VSDT is a CASE Tool, developed to support the idea of Process Oriented Agent Engineering, where Agents are designed by defining use cases and processes described with the BPMN. It's features include the BPMN editor, process structure view, model validation, import of existing web services, transformations to BPEL, and JIAC and many more.

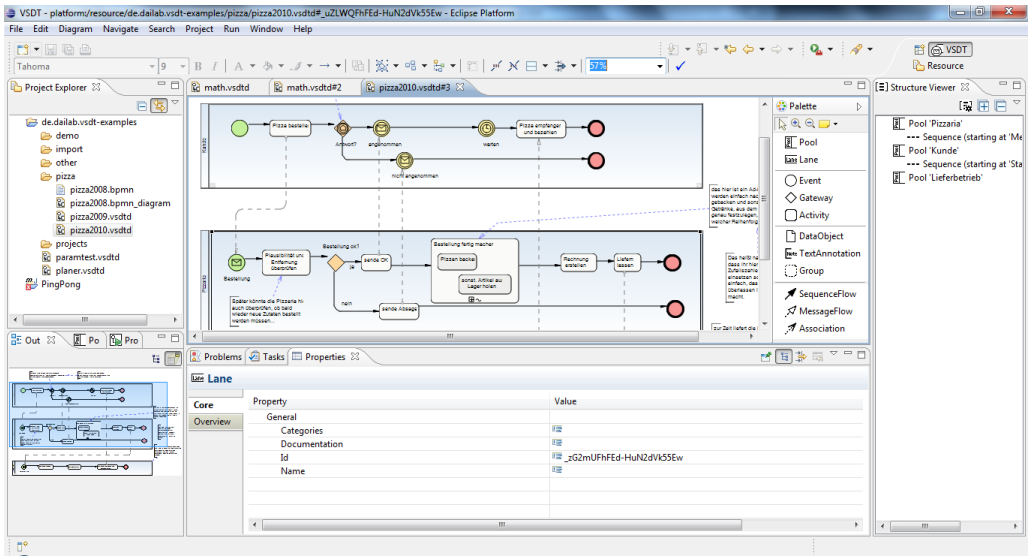


Figure 2.7: VSDT - Editor View

2.3.1 BPMN Editor

The BPMN Editor in VSDT was created using eclipse's GMF (Graphical Modeling Framework). 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.

2.3.2 Transformation Framework

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

1. *Validation*: Validate the input model.
2. *Normalisation*: 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, normalisation and structure mapping are mostly independent from the target language, the standard mapping provided for these stages are reuseable, which makes it possible to implement a new transformation by specifying the element mapping only. Figure 2.8 shows the UML Class Diagram of the transformation famework with the example transformation to BPEL.

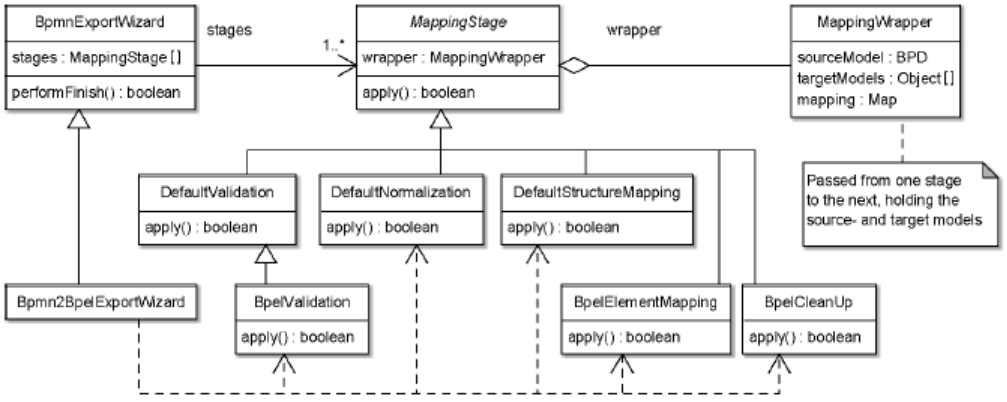


Figure 2.8: Essential classes of the transformation framework, including the BPEL case.[15]

2.4 JET(Java Emitter Templates)

JET[13] is a code generating framework, developed as a part of the Eclipse Modelling Project [7]. Using the so-called templates, one can transform a model into various type of text, from a simple plain text up to text containing html, xml or java code. A naming convention exists 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 it's generate method to get the result String which we can process further for example writing it into a file.

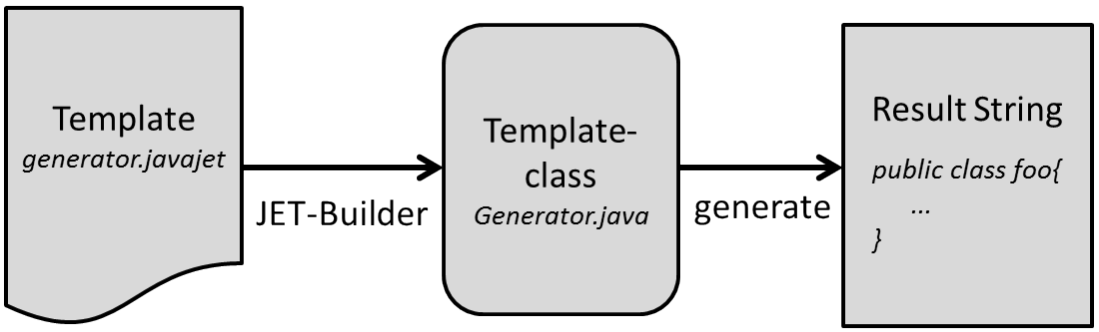


Figure 2.9: JET transformation steps.

2.4.1 JET-Templates

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

1. **Directives** <% *directive* {*attribute* = *"value"*} *%>
   Directives contains 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 it's result will be added into the StringBuffer.

A set of JET templates is called transformation. It is possible to build this transformation with a main template which acts as a visitor and runs through the model, and 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 handles 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 informations 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, 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: 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 2 different JET-Versions

There are currently 2 different JET-Versions in the Eclipse Modelling 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. To directly transform the BPMN directly from it's xml-Representation will be harder and the template will be confusing, therefore a decision has been made to implement

the mapping using the existing transformation framework where an intermediate model class of the AgentBean will be created, and then transform this intermediate model into Java code using the older version of the JET Transformation. More details on the implementation will be discussed in chapter 5.

3. State of the art

3.1 BPMN to BPEL

In the BPMN Specification, the mapping from BPMN to BPEL has been included since the first version. In fact the development of BPMN was driven by the lack of standard notation for the WS-BPEL[19].

For this Project the specified mapping from BPMN2BPEL are used to gain a better understanding of the notation's semantics.

3.2 Existing Transformation to JiacV

At the moment, the VSDT is already equipped with a transformation of BPMN to JiacV, which generates the JADL(Jiac Agent Description Language). The transformation to Jiac AgentBeans 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 we can find in the following table:

Advantages of:	
JADL	AgentBeans
can be deployed to a running jiac application	Written fully in Java, therefore developer friendly. Java is more powerful that JADL. Better performance because no parser are involved.

3.3 WADE

A similar approach (designing agents behaviour with processes and transforming it into Java Code) has been developed by the Telecom Italia with their JADE-extension called WADE (Workflows and Agents Development Environment). 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 executes tasks defined as workflows.

3.3.1 JADE (Java Agent DEvelopment Frameork)

JADE is an application framework and a middleware written in Java, which support the development of software agents. The framework provides distributed runtime environments, agent and behaviour abstractions as well as communications between agents and discovery mechanisms. We can say that it's role is very similar to JIAC.

3.3.2 WADE (Workflows and Agents Development Environment)

WADE is an extension to JADE, which enrich 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 is 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.

3.3.3 Wade-Workflow

A Process in Wade consists a set of Activities. A Process has exactly one **start activity**,

Despite having all the advantages of a Java code, the WADE workflow is rather simple and not so expressive as the BPMN. Similarity to the agent technology such as events and communication flow are also missing in the workflow.

4. Mapping BPMN to Jiac AgentBeans

The element mapping from BPMN to Jiac AgentBeans is created based on the existing mapping to JiacV - JADL script. In comparation to a JADL script, an AgentBean is written completely in Java, enabling more possibilities in mapping concepts such as intermediate Event handling. 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 AgentBean. The name of the AgentBean will be derived from the Pool 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. Further, 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), it may be exposed as an action (for message start event with service implementation), or if the start event is a message start event with MessageChannel implementation, a SpaceObserver will be created and attached to the agent's memory, which will call the workflow method when a message with the payload and address as specified in the implementing message channel are 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** which will be called by the workflow method. Special cases for 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.

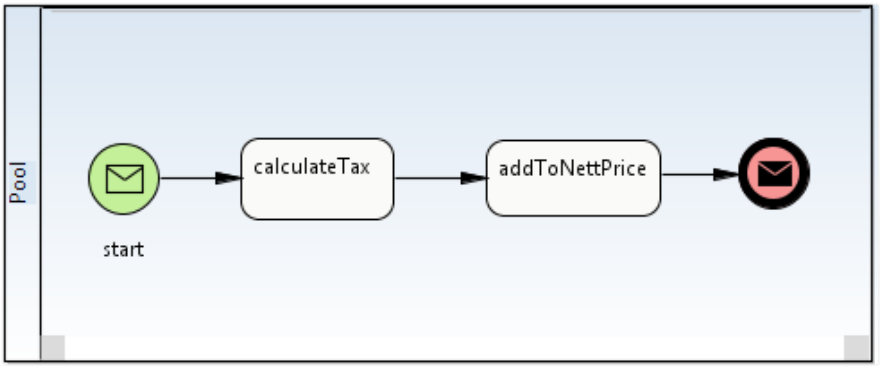


Figure 4.1: Mapping example : Pool and Process

The Figure4.1 shows a simple example of a pool called Pool with the process doProcess. 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.

```
1  package pool;//derived from the pool name
2
3  //some imports...
4
5  public class Accounting_CalculatePrice extends AbstractMethodExposingBean{
6
7    public final static String ACTION_DOPROCESS = "pool.Pool_DoProcess#doProcess";
8
9    // process attribute would be declared here
10   [...]
11   @Expose(name = ACTION_DOPROCESS, scope = ActionScope.GLOBAL)
12   public double doProcess(double nettPrice, double taxRateInPercent) {
13     calculateTax();
14     addToNettPrice();
15     return totalPrice;
16   }
17
18   private void calculateTax(){
19     [...]
20   }
21
22   private void addToNettPrice(){
23     [...]
24   }
25 }
```

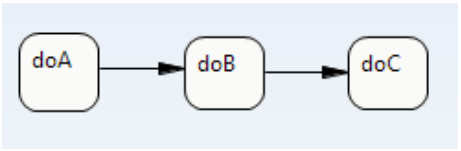
Listing 4.1: Mapped element: Pool and Process (Figure 4.1)

4.2 Workflow Structure

Workflow structure is mapped as 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:



=

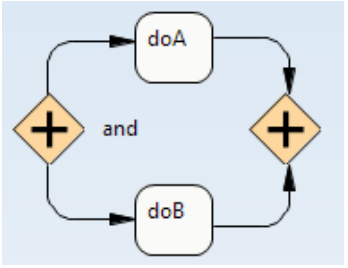
```
doA();
doB();
doC();
```

4.2.2 Gateways

Branches of a gateway are wrapped according to the gateway's type. There are 5 different types of gateways:

1. AND
2. OR
3. XOR_Data
4. XOR_Event
5. Complex

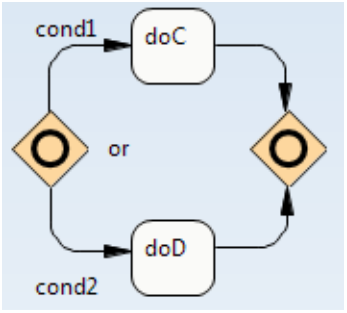
AND-Gateway
In an AND-Gateway, all branch will be wrapped in parallel to one another. At runtime all branches are executed within a thread.



=

```
[...]
Thread and_branch1 = new Thread(){
  public void run(){
    doA();
  }
};
Thread and_branch2 = new Thread(){
  public void run(){
    doB();
  }
};
and_branch1.start();
and_branch2.start();
try {
  and_branch1.join();
  and_branch2.join();
} catch(InterruptedException) { }
[...]
```

(Inclusive)OR-Gateway
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. At runtime, branches for which the condition is not hold will be skipped. However, if it has a branch with default condition, the default branch will only be executed if none of the other branches are executed.

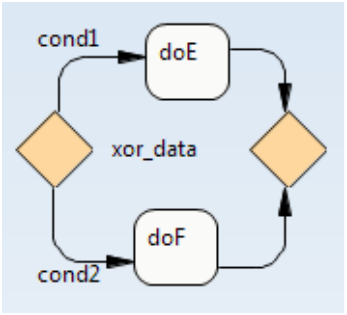


=

```
[...]
Thread and_branch1 = new Thread(){
  public void run(){
    if(cond1){
      doC();
    }
  }
};
Thread and_branch2 = new Thread(){
  public void run(){
    if(cond2){
      doB();
    }
  }
};
and_branch1.start();
and_branch2.start();
try {
  and_branch1.join();
  and_branch2.join();
} catch(InterruptedException) { }
[...]
```

XOR_Data-Gateway

Branches of an XOR_Data are wrapped in an if-then-else block. Only one branch will actually be executed at runtime.

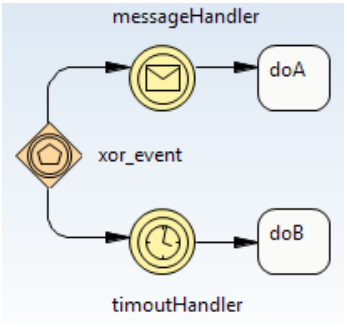


=

```
[...]
if(cond1){
  doE();
} else {
  if(cond2){
    doF();
  }
}
[...]
```

XOR_Event-Gateway

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 recieve an event, the waiting thread will be stopped, and the process continues with the elements of the branch, and other branches will be skipped. The event handler will be added as an inner class to the bean, and they will have a constructor with a Thread toStop argument (among other arguments).



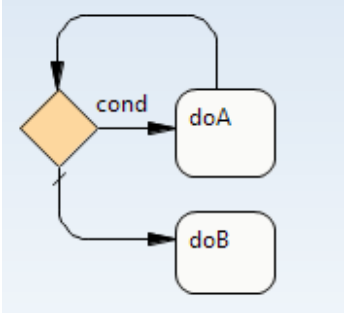
=

```
[...]
Thread xor_event_wait = new Thread(){
  public void run(){
    while(true){
      try{
        Thread.sleep(100);
      } catch(InterruptedException e) { }
    }
  }
};
MessageHandler messageHandler = new Mes-
sageHandler(xor_event_wait);
Timeout timeoutHandler = new Timeout-
Handler(20000, xor_event_wait);
xor_event_wait.start();
messageHandler.start();
timeoutHandler.start();
try{
  xor_event_wait.join();
  timeoutHandler.stop();
  messageHandler.stop();
} catch(InterruptedException e) {
  if(messageHandler.hasBeenTriggered()){
    doA();
  }
  if(timeoutHandler.hasBeenTriggered()){
    doB();
  }
}
[...]
```

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 follows:



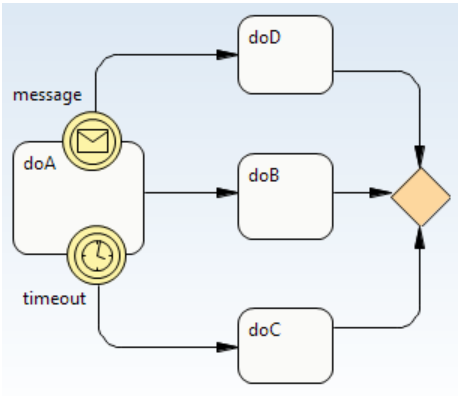
=

```
[...]
while(cond){
  doA();
}
doB();
[...]
```

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

4.2.4 Event Handler

The mapping of an Event Handler attached to an activity is somewhat similar to the mapping of gateway event. 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.



=

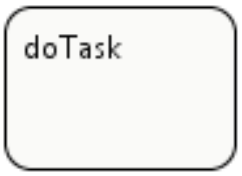
```
[...]
Thread doA_Thread = new Thread(){
  public void run(){
    doA();
  }
};
MessageHandler messageHandler =
new MessageHandler(doA_Thread);
TimeoutHandler timeoutHand-
ler = new TimeoutHandler(20000,
doA_Thread);
doA_Thread.start();
messageHandler.start();
timeoutHandler.start();
try{
  doA_Thread.join();
  timeoutHandler.stop();
  messageHandler.stop();
} catch(InterruptedException e) {
  if(messageHandler.hasBeenTriggered(){
    doD();
  }
  if(timeoutHandler.hasBeenTriggered(){
    doC();
  }
}
if(!messageHandler.hasBeenTriggered()){
  if(!timeoutHandler.hasBeenTriggered()){
    doB();
  }
}
[...]
```

4.3 Activites

Now we will discuss the activities in details. Activities are divided into tasks and subprocess. As mentioned before, 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. Instead a subprocess will be wrapped in an inner class.

4.3.1 Tasks

Basically, a task will be mapped as follows:



=

```
public void doTask(){
  [declaration of properties]
  [start assignments]
  [special mapping depending on task-type]
  [end assignments]
}
```

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). 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*.

Service-task

Send-task

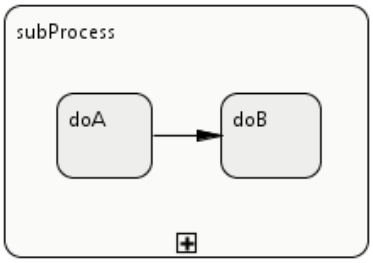
Recieve-task

Other-tasks (currently not mapped)

4.3.2 Subprocess

A subprocess will be mapped into an inner class of the containing Process or Subprocess. This way it's 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:



=

```
public void doProcess{//workflow-method
  [...]
  SubProcess subProcess = new SubPro-
cess();
  subProcess.run();
  [...]
}
[...]
class SubProcess {
  [declaration of properties]
  public void run(){
    doA();
    doB();
  }
  private void doA(){ [...] }
  private void doB(){ [...] }
}
```

4.3.3 Activity-Looping

Standard-Loop

Multi-Instance Loop

4.4 Events

4.4.1 Start-Events

Timer

Message

4.4.2 Intermediate-Events

Timer

Message

4.4.3 End-Events

5. Implementation of the transformation

In this chapter, some details of the transformation implementation will be presented.

5.1 Transformation Stucture

As mentioned before in section 2.3, the transformation process in the VSDT is divided into 5 stages. We've also mentioned that the default validation and structure mapping provided by the transformation framework are reuseable. For the implementation of the new transformation, the framework's DefaultBpmnValidator and BPMN2StrucBPMNTransformation are being reused, as we can see in Figure 5.1.

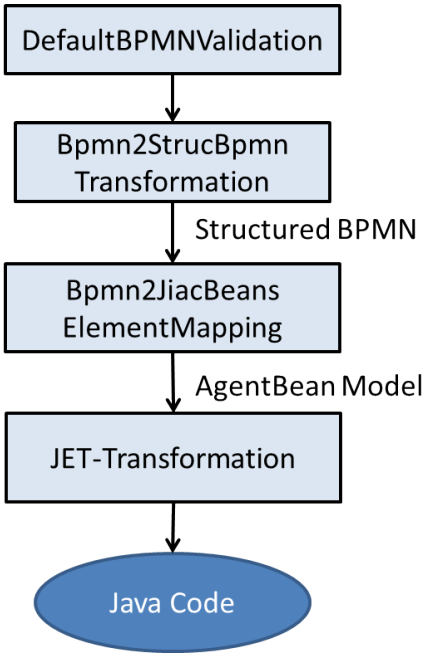


Figure 5.1: Transformation stages

The Element mapping stage is implemented on the basis of the existing Bpmn2JiacV-ElementMapping. It takes a structured Bpmn as a model and transforms each pool

contained in the model's business process diagram into a java object, which represent a model of a java file holding an AgentBean class. For this implementation, a Metamodel of the Jiac AgentBean(see Figure 5.2) was developed as an intermediate product.

After all Pools in the business process is completely visited and transformed into AgentBean models, these models are passed over to the JET-Transformation where they will be transformed into a String which represents the content of a Java File.

5.2 AgentBean Model

An AgentBean model has a list of attributes, methods, action and it may also have some subprocess(because a subprocess is mapped into an inner class of the generated AgentBean).

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.

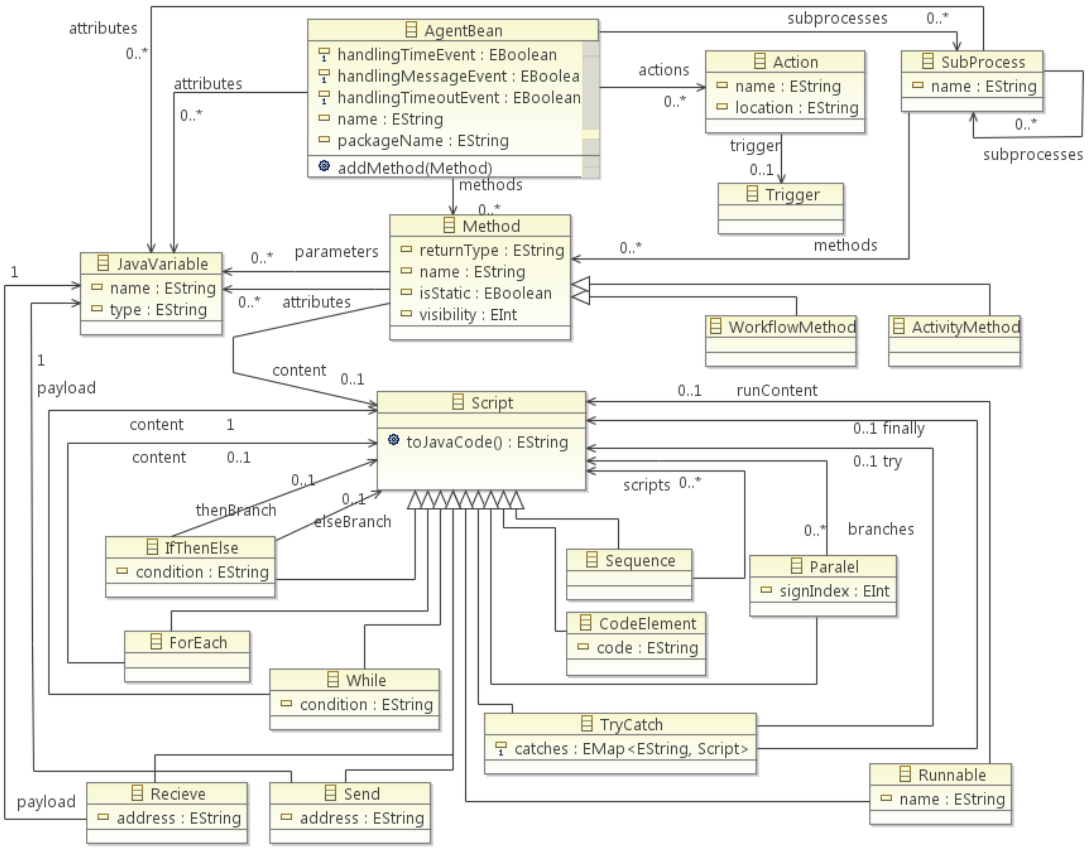


Figure 5.2: AgentBean - Metamodel

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:

```
1  public String toJavaCode() {
2    String code = "";
3    code += "if("+condition+"){\n";
4    if(thenBranch!=null){
5      BufferedReader reader = new BufferedReader(new StringReader(thenBranch.
           toJavaCode()));
6      try{
7        String line = reader.readLine();
8        while(line!=null){
9          if(!line.equals("")) code += "\t"+line+"\n";
10         line = reader.readLine();
11       }
12     }catch(IOException e){
13       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 this method will be used by the JET-Transformation and the result will then be written in a *.java File. Therefore, as you can see in line 7-11 and 21-25, a tab are added in front of each line in the then and else branch.

The role of MDE in the Implementation

The benefits of Model Driven Engineering are also found during the implementation of the AgentBean model. As we can see in Figure 5.2, it is created graphically using eclipse's Ecore Tools - Ecore Diagramm. With the help of the EMF Generator each element of the model can be easily generated into JavaCode 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.

5.3 JET-Transformation

The JET-Transformation consists of 4 Templates(see also Figure 5.3):

1. agentbeantemplate.javajet
2. subprocesstemplate.javajet
3. timeouteventhandler.javajet
4. messageeventhandler.javajet

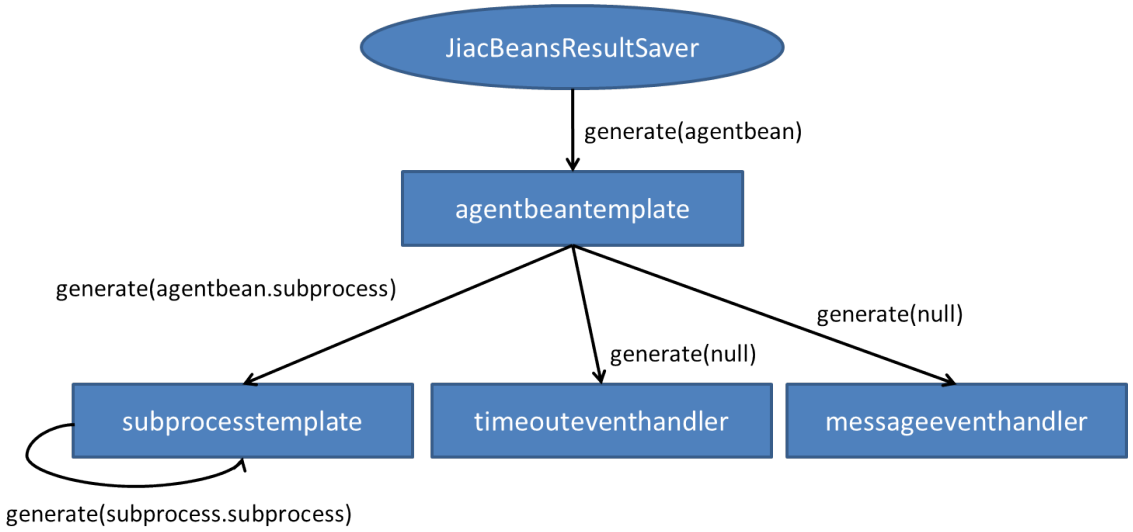


Figure 5.3: JET-Transformation Structure

The agentbeantemplate is the main template of the implemented JET-Transformation. An instance of it's Java template class are created by the JiacBeansResultSaver and the generate method will be invoked for each AgentBean model generated in the element mapping stage.

The subprocesstemplate is invoked by the agentbeantemplate and recursively by the subprocesstemplate itself for each subprocess(if any exists) contained it's argument(an agentbean model or a subprocess model).

Both timeouteventhandler and messageeventhandler are static templates, which means the result of the generate method does not depend on the argument object. They simply add an inner class TimeoutEventHandler or MessageEventHandler to the AgentBean. They are invoked by the agentbeantemplate if the value of the flag handlingTimeoutEvent or handlingMessageEvent of the agent bean model is true.

5.4 Merging generated and manually edited code

One of the 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**[12]

5.4.1 JMerge

5.5 Open Issues

implementation not complete

using JET2

better mergerules to mix manually edited and generated code within a method

6. Examples

In this chapter we can see a selection of BPMN-examples and how the resulting Code of the transformation look like.

6.1 Simple Flow

6.2 Time Event Handler

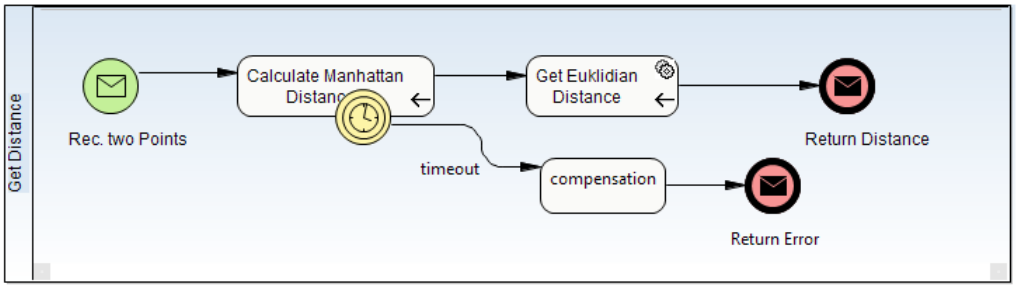


Figure 6.1: Process with event handler

7. Conclusion

7.1 Future Work

Complete the Implementation of existing Mapping

Literaturverzeichnis

[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 Heer. Merging agents and services the jiac agent platform. In Amal El Fallah Seghrouchni, Jrgen 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] JADE Homepage. `http://jade.tilab.com/index.html`.

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

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

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

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

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

[14] 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.

[15] 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.

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

[17] 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.

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

[19] 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.