ulm university universität
uulm

Ulm University | 89069 Ulm | Germany

Faculty of Engineering and Computer Science
Institute of Databases and Information Systems

Natural Language-based Visualization and Modeling for Updatable Process Views

Bachelor Thesis at Ulm University

Submitted by:
Wolfgang Wipp
wolfgang.wipp@uni-ulm.de

Reviewer:
Dr. Manfred Reichert

Supervisor:
Jens Kolb

2013

Edition July 30, 2013

Satz: PDF-LaTeX 2$_\varepsilon$

Abstract

Nowadays, an understanding of its own business processes is crucial for companies, to ensure an efficient and quick changing work flow.

While several tools exists using graphical annotations, e.g. Business Process Model and Annotation (BPMN), an untrained staff member may not be able to understand business processes described in these annotations, resulting in additional investments into staff member training. Furthermore, some structures used in graphical annotations may not seem native to untrained users, causing misinterpretations of business process models. Fostering this issue, natural language-based process descriptions may used. These descriptions are automatic-generated from process models.

Another problem of modern business process modeling is the communication between process modelers and domain experts. A thought of a domain expert can be misinterpreted by the process modeler. The results are discrepancies in business process models. Avoiding this problem, modeling mechanics for natural language-based process descriptions may used.

Therefore, the thesis introduces fundamentals of the proView project as well as the generation and editing of natural language-based process descriptions. Subsequently, integration steps of natural language into the proView project, divided into two parts, are shown. The first part contains integration of a process model to natural language text converter. The second part discusses problems and solutions of natural language-based modeling. Afterwards, the second part shows the integration of natural language-based modeling into the proView project.

Finally, further steps in the future are discussed.

Acknowledgements

First of all, I would like to thank Jens Kolb for his support and important guidance through all of this thesis. Additionally, I would like to thank Henrik Leopold for his permission to use parts of his prototype of a process to text converter.

My deepest appreciation goes to Michael, Bernd, Steini, Raphael, and Christoph, my fellow students, for their moral support and advice.

A very special thanks goes to Stefan Büringer, who helped me with my endless fight against the Vaadin framework and was never too busy to answer my questions about the proView project.

Last but not least, my thanks goes to my parents Heinz and Brigitte, who believed in me the whole time, not only while writing this thesis, but all of my life. Without you, this thesis would still be a mist in my thoughts.

Contents

1

Introduction

In modern business environments, flexibility and quick reactions to market changes are requirements with increasing importance, to sustain on the world market. To be flexible and quick, all business processes of the company have to be documented and possibly re-engineered.

Documentation of business processes is done with the help of process participants. These process participants are interviewed by a process modeler, resulting in a user story. Then, the process modeler creates a first business process model based on these user stories. Afterwards, each process participant reviews the business process model. This step of modeling and reviewing iterates until the business process is documented correctly.

A documentation itself can be done in different annotations. Most of annotations are graph-based, such as *Business Process Model and Notation* (*BPMN*) or *Event-driven Process Chains* (*EPC*).

However, most process participants are not used to graph-based annotations. This might lead to misinterpretations of annotation-specific structures, that haven't got native understanding, resulting in misunderstanding of respective processes. Furthermore, even simple concepts like the logical 'or' and 'xor' are easily misunderstood by non-technical process participants, interpreting a logical 'or' as 'xor'. This leads to the fact, that a company must train their staff members in process modeling, or at least in the used annotation. However, training their staff costs companies not only money, but time as well.

This leads to the fact, that a solution with natural language-based and graph-based business process descriptions has to be found. Having a solution, that uses one central business process model with different kinds of visualizations i.e., a natural language-based text and e.g. BPMN, helps both process modelers and process participants. A process modeler can still model a process graph, while a process participant can read a textual description of the process. With this, costs, in terms of time and money, for training process participants in the used annotation are reduced. Furthermore, the 'language barrier' between the process modeler and the process participant can be decreased.

However, if a process participant wants to model changes in a business process, she needs to use graph-based annotations. To create a two-way communication over business process models, natural language texts should be editable in a way, that affects not just the text, but the business process model. With this ability, process participants can read natural language texts and model changes in a business process, without the need of graph-based business process modeling experience.

Therefore, the thesis is structured as follows: Section 2 introduces fundamentals of process modeling and proView. Subsequently, Section 3 discusses the integration of natural language generation in the proView prototype. Section 4 introduces process

modeling based on natural language descriptions. Section 5 exposes the implementation of the natural language-based modeling component of proView.

Finally, Section 6 concludes and discusses further steps in the proView projects natural language-based modeling functionalities.

2

Fundamentals

This section introduces the fundamentals for this bachelor thesis. Section 2.1 shows the proView project. Section 2.2 describes the process of natural language-based business process description generation. Finally, Section 2.3 gives a small introduction in the fundamentals of HTML and Java script.

2.1. Fundamentals of proView

This section introduces the proView project. Section 2.1.1 shows the concept of Central Process Models and process views. Section 2.1.2 introduces the proView project, along with the proView framework.

2.1.1. Central Process Models and Process Views

A *Central Process Model (CPM)* is the process model that comprises all information concerning a respective business process. In particular, it consists of follows elements [1]:

- **start event:** A start event is the entry point of a process model. It has only outgoing control edges.
- **end event:** An end event is the point of process termination. It has only incoming control edges.
- **activity:** Activities are steps in the business process, where each activity represents a step or multiple steps (sub-process).
- **gateway:** Gateways split or join the control flow of a process model. Therefore, different types of gateways can be used with different semantic. For example:
  - **AND**: All following *branches* are executed. A branch is a control flow from one splitting gateway to the corresponding joining gateway.
  - **OR**: At least one of the following branches gets activated.
  - **XOR**: Exactly one of the following branches gets activated.
- **control edges:** Control edges lead the control flow from activity/gateway to another activity/gateway from the start to the end of the process model.
- **data element:** A data element holds data of a specific type, such as string, integer, or personalized data types. Data elements can be read by activities and gateways as well as wrote by activities.
- **data edge:** A data edge shows the relation between an activity/gateway and a data element. It shows which element reads or writes a data element.

However, a CPM may be very large and complex. Unfortunately, the human brain is only capable of understanding a limited complexity in general [2]. Therefore, it is important to reduce the complexity of such a CPM to support users in understanding them.

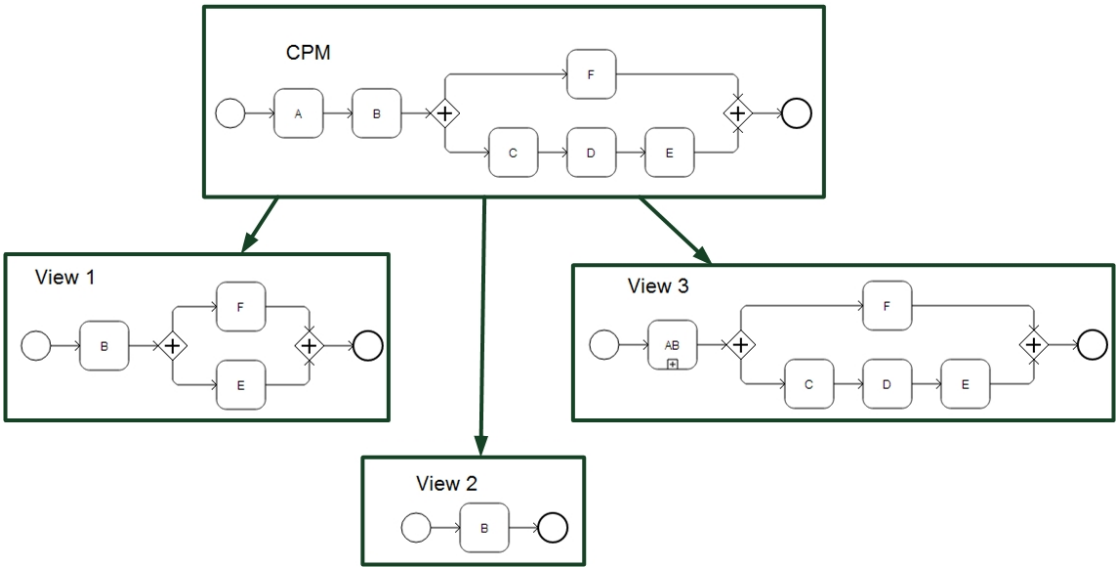

Figure 2.1.: CPM and Corresponding Process Views

Sub-processes reduce the number of elements and still show everything that is important for a user. However, the user may need only specific information (e.g. only tasks affecting her), yet, she is stil confronted with a lot of non-relevant information.

Therefore, a *process view* offers a customized version of a CPM. To create a process view, activities might be reduced i.e. respective activities are hidden in the view, but still existing in the corresponding CPM. Additionally, multiple activities might be aggregated to one abstract activity. These operations are called *view create operations* [3], shown in Table 2.2.

Through the possibility of creating multiple process views based on a CPM, customized and personalized process views for each process participant can be created. An example CPM and process views are shown in Figure 2.1. In *View 1*, activities 'A', 'C', and 'D' are reduced. In *View 2*, all activities and gateways except for activity 'B' are reduced. In *View 3*, activities 'A' and 'B' are aggregated to an abstract activity 'AB'.

Additionally to view create operations, update operations to modify corresponding CPMs, called *view update operations* shown in Table 2.1 are supported [3].

Operation	Action
Delete Element	Delete Selected Element
Add Activity	Add New Activity
Update Activity	Rename Activitylabel
Add Gateway	Add New Gateway
Add Data Element	Add New Data Element
Write Data Element	Selected Activity writes Selected Data Element
Read Data Element	Selected Activity reads Selected Data Element
Update Data Element	Change Properties like Type or Name of Data Element

Table 2.1.: View Update Operations

Operation	Action
Reduce Activity(-ies)	Hide Activity in Process View
Aggregate Activites	Combine Selected Activites to one Abstract Activity
Create View From Selection	Create a New Process View With Selected Elements in it
Show Subprocess	Show Process which is in an Abstract Activity or Subprocess
Reduce Data Element(s)	Hide Data Element in Process View
Aggregate Data Element(s)	Combine Selected Data Elements to one Abstract Data Element

Table 2.2.: View Create Operations

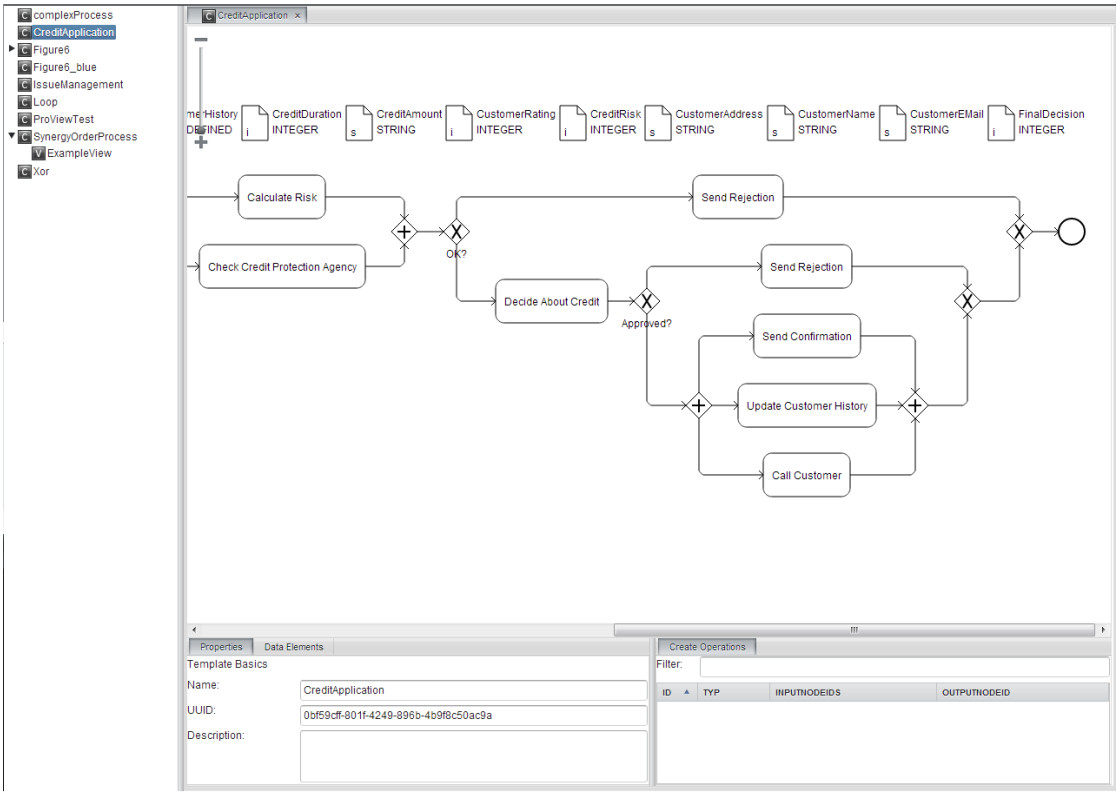

Figure 2.2.: The proViewClient

To ensure a consistent CPM/process view relation, each process view has a *creation set*, which consists of the required view create operations to rebuild the respective view from the corresponding CPM [3].

2.1.2. The proView Project

The *proView project* is a research project, which develops the *proView framework* and has a proof-of-concept implementation - the *proView prototype*.

The proView project makes use of CPMs and process views, to ensure understandability of large business processes.

Part of the proView prototype is implemented by Stefan Büringer during his bachelor thesis [1]. It consists of the *proViewClient*, a Vaadin-based web application [1], shown

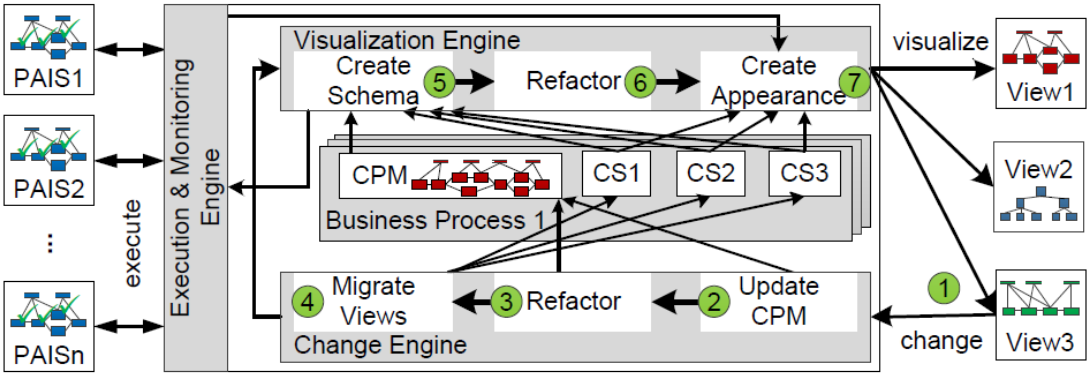

Figure 2.3.: The proView Framework [5]

in Figure 2.2. Furthermore, there are other clients, e.g. an iPad/iPhone application for multi-touch gesture modeling [4].

The proViewClient communicates over a *REST interface* with the *proViewServer*. The proViewServer handles the business process models, which means that it stores the business process models and performs operations on them [5]. An overview over the implemented proView framework is shown in Figure 2.3.

A possible operation on a process view is executed as follows [5]:

- **Step 1:** After a user updates a process view, the *change engine* is triggered.
- **Step 2:**Then, the process view change is used on the corresponding CPM.
- **Step 3:**Afterwards, the CPM gets simplified, e.g. no more needed elements are removed.
- **Step 4:**The update is used on all corresponding process views creation sets.
- **Step 5-7:** All process views are recreated.

Afterwards, the updated process views can be visualized in the proViewClient.

The representation of CPMs and process views in the proView prototype is done by the class *Template*. It consists of *Node*s(i.e., activities, gateways, and data elements) and *StructuredEdge*s(i.e. control edges and data edges). Furthermore, if the template represents a process view, it holds the *ViewCreateOperation*s which create the process

view out of the respective CPM. A Node has an type attribute that shows what type of a process model it is. StructuredEdges have an attribute to decide if they are a control edge, a data edge or an edge for a loop structure. Furhtermore, the StructuredEdges hold the ids of the respective elements they connect.

2.2. Fundamentals on Generating Natural Language Texts from Business Process Models

This section introduces the fundamentals of generating natural language texts from business process models. The principles described are from the Ph.D. thesis of Henrik Leopold [6].

Section 2.2.1 gives an overview over the act of transformation. Section 2.2.2 shows the first of three steps, the text planning. Section 2.2.3 shows the second step, the sentence planning. Section 2.2.4 shows the third and last step, the surface realization. Finally, Section 2.2.5 introduces the implemented prototype.

2.2.1. Overview: The Process of Generating Natural Language Texts from Business Process Models

This section gives an overview over the act of generating a natural language text from a business process model and discusses the given challenges in each step of the generation process.

The task of process model to natural language text is split into three main steps, i.e., *Text Planning*, *Sentence Planning*, and *Surface Realization* [7].

The first two steps are split up into sub steps as shown in Figure 2.4. The first step plans the structure of the text, using previously extracted informations, i.e., business process model structure, from the business process model [7]. Step two plans the sentences to be made and refines sentence structures [7]. Finally, step three generates the planned sentences [7].

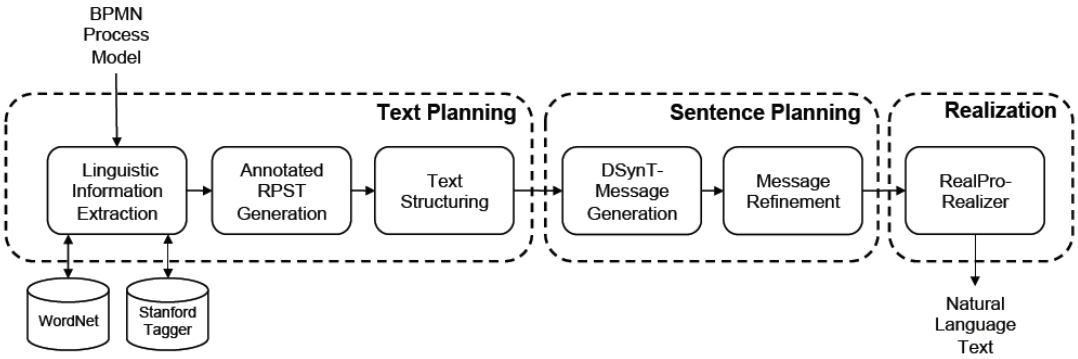

Figure 2.4.: Natural Language Generation Process [7]

2.2.2. Step 1: Text Planning

A given process model is analyzed by its activity labels and its structure. First, it uses WordNet and the Stanford Parser for activity label analysis [7]. Then, a *Refined Process Structure Tree* (*RPST*) [8], by applying a modified RPST generation algorithm [7], of the business process model is generated.

The combination of *WordNet* and the *Stanford Parser* allows to detect different labeling styles and extract the *action* and *object* of an activity label as well as *modifiers* [7].

WordNet is a large lexical database of English nouns, verbs, adjectives, and adverbs [9]. WordNet provides semantical relations between words, e.g., 'dog' is related to 'animal or 'waiter' is related to 'human' [9]. WordNet consist of so called *Syn-sets*, where each Syn-set is a combination of multiple synonyms, having somehow a relation with each other [9]. For example: a dog, a cat, and a mouse would be in the Syn-set 'animal' for being animals.

The Stanford Parser is a natural language parser, capable of analyzing sentence structures. The parser can mark words according to their syntactical types in sentences [10]. The marks of basic word types are as follows [11]:

- **NN**: Noun, singular or mass
- **VB**: Verb, base form

- **JJ**: Adjective
- **RB**: Adverb

The whole set of tags is shown in the context of building the *Penn Treebank* [11].

To increase readability, the generation approach makes use of bullet points and paragraphs. For this, special parameters have to be set in this step, determining if a bullet point has to be set or how long a paragraph should be [7]. A bullet point, for example, is used when the control flow of a business process is split into multiple branches, where each branch is represented by a bullet point [7]. Furthermore, nested splits of a control flow are saved in the parameters as well, using a *level* attribute [7].

2.2.3. Step 2: Sentence Planning

A *Deep Syntactic Tree* (*DSynT*) is a tree-based dependency representation introduced in the context of the Meaning Text Theory [12]. A DSynTs node carries a semantically full *lexeme*, i.e., a semantic composition of words, ignoring their inflection [7]. Each lexeme carries grammatical meta information as for verbs, the tense or number, or for nouns, the definiteness, called *grammemes* [7]. Therefore, DSynTs are perfect for powerful, but manageable, sentence representations. An example DSynT of the sentence 'The room-service manager takes down the order.' is shown in Figure 2.5.

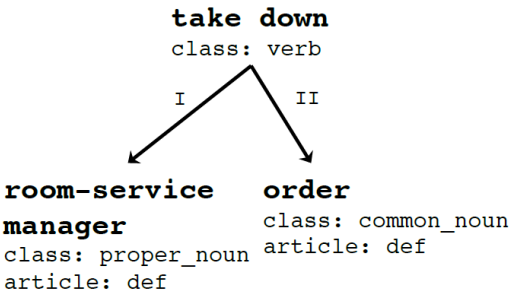

Figure 2.5.: A simple DSynT [7]

In this step, a DSynT for each activity in the business process model is generated out of the extracted information from the activity label and the given business process model. The generated DSynTs are ordered in a list with the help of information given by the previously generated RPST [7].

Additionally, this step adds translations from process structures into natural language-based text parts. For example: An AND-gateway split is represented in the text with a phrase like: 'The process is split into [number of branches] branches' [7].

Finally, *phrase aggregations*, *referring expressions*, and *discourse markers* are integrated in the text to improve textual quality [7]. Their meaning is as follows [7]:

- **Phrase Aggregation:** A phrase aggregation means the combination of two sentences to create a new sentence that holds the same information.
- **Adding of Referring Expressions:** Pronouns, i.e., she, he, or it, are set in the text, replacing repetitive occurrences of nouns.
- **Insertion of Discourse Markers:** Conjunctions, such as *afterwards* or *subsequently* are added to the beginning of sentences for better readability.

2.2.4. Step 3: Surface Realization

In this last step, grammatically correct sentences are produced. This requires to find a suitable word order, a correct inflection of verbs, punctuation and capitalization as well as introduction of function words [7].

As result of the whole generation process, a natural language text, describing the input business process model, is generated.

2.2.5. The ProcessToTextTransformer Prototype

The *ProcessToTextTransformation* (*PTTT*) prototype, a Java application, is created by Henrik Leopold and Sergey Smirnov. It translates BPMN process models into natural

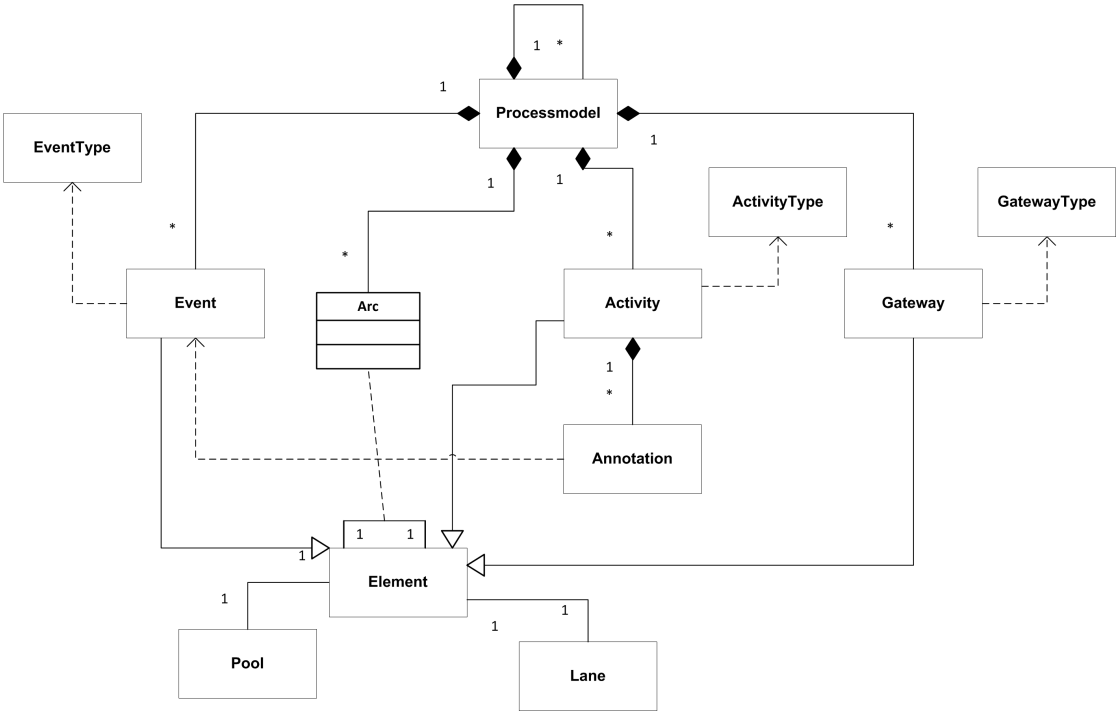

Figure 2.6.: PTTT Process Structure

language text, using RPSTs and DSynTs. Furthermore, it makes use of WordNet and the Stanford parser. The execution of translation is as described in the previous sections.

The PTTT component uses Java class-based data structures to represent business process models. An overview over the classes used for business process representation is shown in 2.6.

A *ProcessModel* holds the *Events*, *Arcs* (control flow edges), *Activities*, and *Gateways* of the business process model. Furthermore, it can have multiple process parts from type ProcessModel. This is required because of the possibility of disrupted control flow by events.

Events, Activities, and Gateways inherit the class *Element*, which holds the *Lane* and *Pool* the Element is in. An Activity can hold multiple *Annotations*, actions and business objects descriptions, of the Activity. Activities, Gateways and Events have according *Type* classes. An Arc holds a *source* and *target* Element.

The three steps are represented by three classes with the respective steps name, e.g., the surface realization step is done by the class *SurfaceRealizer*.

For the step of surface realization, the publicly available realizer named *RealPro* from *CoGenTex* is used, that allows to generate grammatically correct sentences out of XML-based DSynTs [13]. Therefore, the problems mentioned in Section 2.2.4 are solved. The result is a string, using tabs and line-breaks for structuring [7].

2.3. Fundamentals of HTML and Java Script

This section gives a small introduction of HTML and Java script, since both langauges are used to create a custom component, described in Section 5.1. Therefore, Section 2.3.1 describes the principles of HTML. Finally, Section 2.3.2 describes the principles of Java script.

2.3.1. HTML

HTML is a mark-up language for internet websites. The newest version, HTML5, is a cooperation between the World Wide Web Consortium and the Web Hypertext Application Technology Working Group [14]. To create a new website, a programmer writes a new HTML document that consists of the structural parts of the website, e.g., headlines, navigation area, footer, and header. Therefore, several elements exist with different structural semantic. In this thesis, the used elements are:

- ** :** This is a paragraph.
- **:** This is a section.
- **<ul>:** This is an unordered list.
- **<li>:** This is an item in an unordered list.
- ** :** This is also a section in a document.

Such elements are called *tags*. Each tag consists of two types: an opening (e.g.,) and a closing (e.g.,) tag. Furthermore, tags can be nested. So the nesting '

  ' is allowed. However, an opened nested tag must close before the tag in that its nested. Therefore, '   ' is not allowed.

Every tag has attributes. The used attributes in this thesis are:

- **id:** This is an attribute that holds an unique identifier string.
- **onclick:** This attribute can link to Java script functions, when the element is clicked with the left mouse button.
- **oncontextmenu:**This attribute can link to Java script functions, when the element is clicked with the right mouse button.
- **contentEditable:** This attribute shows, if the content of an element is allowed to be edited. It is of type boolean.

For example, a div element with the attribute contentEditable set 'true' looks as follows:

' ... '

For further features of HTML, please visit the website of W3C that introduces HTML [14].

2.3.2. Java script

Java script is a light weight scripting language. It can be inserted into HTML pages and executed by all modern web browsers [15]. Java script can manipulate HTML documents at runtime. For this, there are several functions (functions are the equivalent to methods in Java). An example for Java script is shown in Section A.3. Functions are created as follows (source code line 11): 'getId = function(){return id;};'

'getId' is the name of the function, then the word 'function' indicates that we want a new function. In the brackets after function, one can say what parameters this function shall have. Important: Java script has no strict type system. For example, in a variable can first hold an integer and afterwards a string. In the swung brackets is the code to be executed after function call. In this case, the value of an id variable is returned.

With the function 'getElementsByTagName(tagName)', an array of all elements in the HTML document with the specific tag name is returned. Line 34 shows an call of this

function for a div element and then gets the first one by pointing of first element of the array with ’[0]’.

If an element is hold in a variable, one can directly access the elements attributes by the point operator. An example is given in line 37, where the div elements attribute ’onclick’ is set to a string with one blank. For further Java script feature, please visit the website of W3C’s JavaScript tutorial [15]

3

Integration of ProcessToTextTransformer into proView

In the following the generation of natural language process descriptions, described in Section 2.2, through the integration of parts of the PTTT prototype, described in Section 2.2.5, in the proView prototype, is addressed.

At the end of this section, it shall be possible to generate a natural language process description out of CPMs or process views in the proViewClient, view them in a new made Appearance, and export the natural language process description as a PDF file.

Therefore, Section 3.1 introduces the graphical user interface for the natural language process description. Section 3.2 describes the new packages added to the proViewClient. Section 3.3 shows the class level integration step. Section 3.4 describes the changes made on code level in both the proViewClient as well as the PTTT prototype. Finally,

Section 3.5 introduces personalization features of the PTTT prototype combined with the proViewClient.

3.1. Design of the User Interface

Since the proView framework allows to use different appearances for the visualization processes, it is required to create a new appearance for the natural language-based visualization concept. This appearance is called *NLAppearance*, where as 'NL' stands for **N**atural **L**anugage, shown in Figure 3.1.

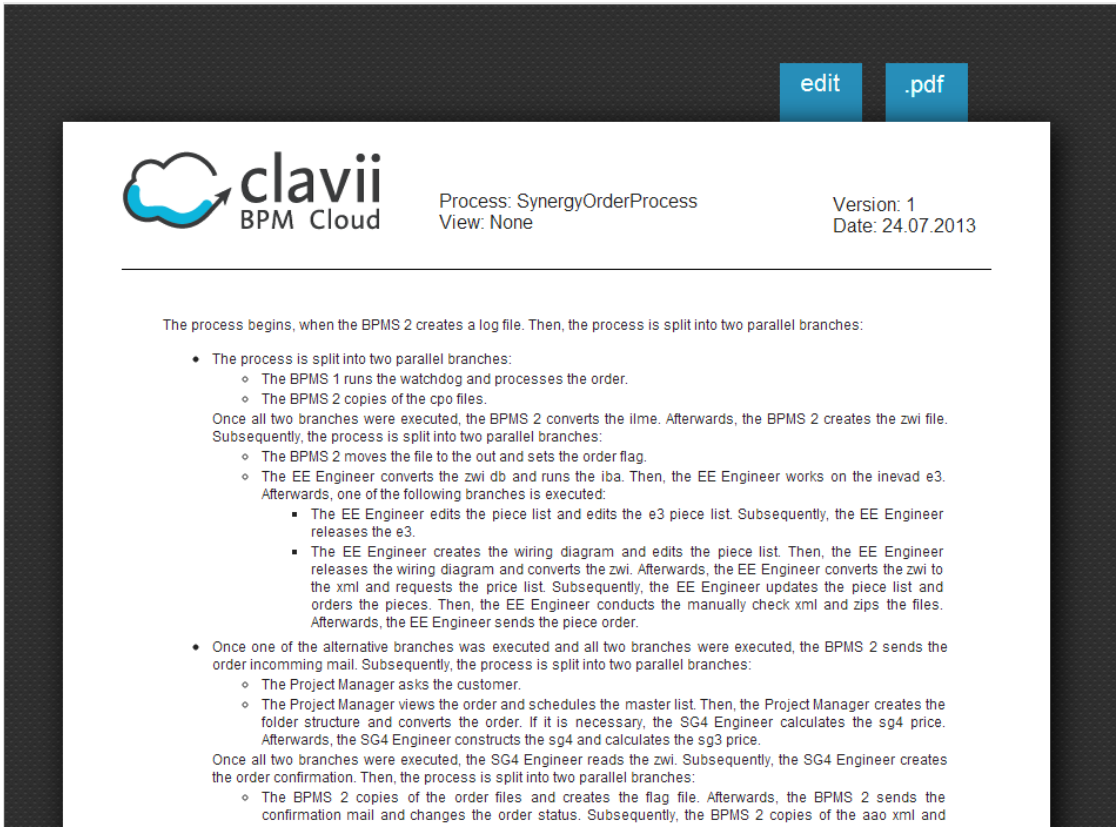

Figure 3.1.: NLAppearance GUI

The goal of the user interface is to simulate a paper sheet to present the natural language process descriptions to the user. Furthermore, the header of this paper sheet would

include the logo of the respective company. The header also contains informations like the name of the process, the name of the view, if a view is visualized, the version of the process and the current date. Sticky notes attached to the paper sheet are used for user interaction with the system , e.g., switching between an view and an edit mode or export the process description as a PDF file, replacing an classical menu structure.

3.2. Package Level Integration

A package for the PTTT component is created, called 'de.unium.proView.wi.utils.processToText', to maintain all classes of the PTTT component in the original structure. However, classes for RPST generation are in a package called 'de.hpi.bpt'. Furthermore, a package for the natural language appearance is created, called 'de.uniulm.proView.wi.appearances.TEXT'. An overview of the proView prototype packages, involved in the integration of the PTTT component in the proViewClient, is shown in Figure 3.2. The new packages are highlighted with red squares.

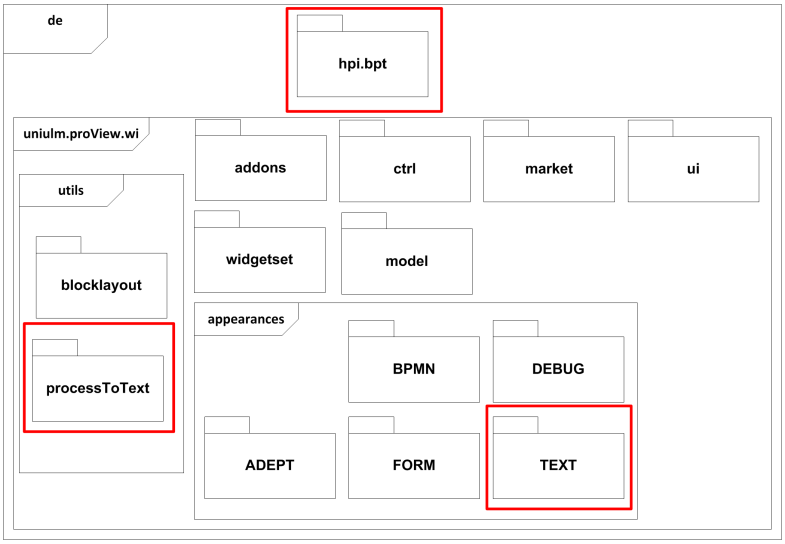

Figure 3.2.: The proView Prototype Package Structure

3.3. Class Level Integration

With the information from Section 2.2.5 and Section 2.1.1, the need for a process translation from the proView prototype template to the PTTT process structure originates. Therefore, a new class is created, called *TempToNLModelConverter*, in the PTTT prototype. The translation is described in Section 3.4.3. Furthermore, the class *NLAppearance*, the user interface class, described in Section 3.1, is created.

The instantiation of the PTTT prototype, with its main class *ProcessToTextConverter*, is done at the start of a new session, due to performance issues on start up. These issues results of the use of WordNet and the Stanford parser, which instantiation take a lot of time, reading their respective configuration file.

The class, that instantiates the ProcessToTextConverter class, is called *AppearanceService*. It provides useful operations and values for appearances. Therefore, the NLAppearance is connected with the AppearanceService. As shown in Figure 3.3, multiple NLAppearances use one instance of ProcessToTextConverter.

The GUI elements of the NLAppearance are multiple *Label* objects for the header of the document, two *Embedded* objects as representation of the sticky notes and one for a company logo in the header, and a Java script component called *NLTextArea*, described in Section 5.1, that holds the natural language process description. The proViewClient can automatic recognize appearances. Therefore, it investigates the 'appearances' package for classes that inherits the abstract class *AAppearance*. Because of this recognition system, the NLAppearance inherits from AApearance.

The class *ModellingService* provides the process operations to appearances. The classes *HTMLParser* and *PDFExporter* are described in Section 3.5. The class *SelectionHandler* is presented in Section 5.2, the class *TextInvestigator* in Section 5.3. The class diagram of NLAppearance is shown in Figure 3.3.

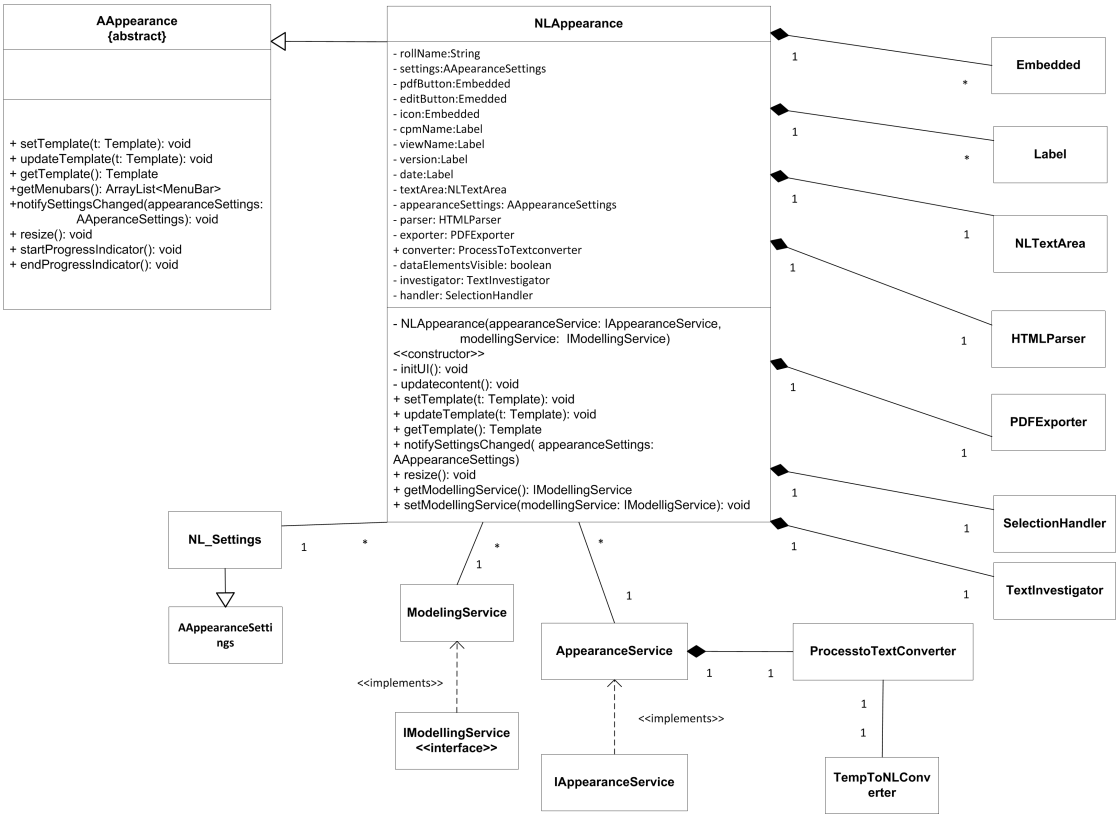

Figure 3.3.: NLAppearance Class Diagram

3.4. Code Level Integration

This section describes the changes of classes, both in the PTTT and the proViewClient, made for the PTTT component integration in the proViewClient.

Section 3.4.1 describes the changes in classes of the proViewClient. Section 3.4.2 shows the changes in the PTTT prototype's classes. Finally, Section 3.4.3 introduces the algorithm for proView process model to PTTT process structure.

3.4.1. Changes in Classes of the proViewClient

The classes changed for PTTT prototype integration are *AppearanceService* and *ProViewWI*, the entry point class of the proViewClient. In the AppearanceService, a new attribute is created for the ProcesstoTextConverter class. Additionally, a *String* type attribute is created for the agent's name, logged in this session. A nested class called *InitConverterThread*, which inherits the class *TimerTask*, is made in ApearanceService, to call it in the instantiation process of proViewClient, having a decoupled thread for the PTTT instantiation.

Because of the need for reading the configuration files of WordNet and Stanford parser, ProViewWI has a new *String* attribute, that holds the base directory of the sessions instance.

The involved classes in ProcessToTextConverter instantiation are as follows:

- **VaadinRequest:** A generic request to the Server [16].
- **ProViewWI:** Inherits class UI, entry point of the session [16].
- **TimerTask:** A task that can be scheduled for one-time or repeat execution [17].
- **LoadingInterfaceTask:** A nested class in ProViewWI, inherits TimerTask.
- **AppearanceService:** A class that provides generic operations for appearances, e.g. *getAvailableHeight*, a method that returns the available height in the browser window.
- **InitConverterThread:** Nested class in AppearanceService, inherits TimerTask. This task instantiates the ProcesstoTextConverter class.

The instantiation process is shown in Figure 3.4. First, a VaadinRequest comes to the server, starting a new session of proViewClient, instantiating ProViewWI [16]. Then ProViewWI starts its *init* method. First the init method determines the base directory of the session with the help of VaadinRequest. Then, the init method calls the method *initLogin*. In this method, a *LoginWindow* is instantiated, a window where the user can log in. Futhermore, at the end of the method, it starts the LoadingInterfaceTask.

There, the Method *initCtrl* from ProViewWI is called. In initCtrl, a new instance of AppearanceService is made.

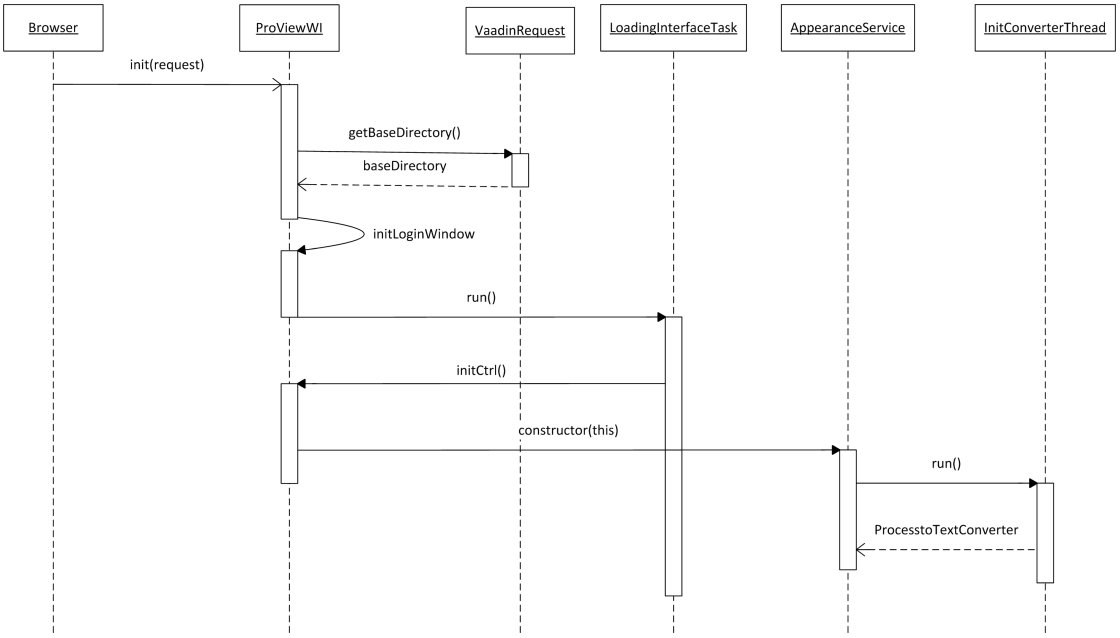

Figure 3.4.: Instantiation of ProcesstoTextConverter at Start of proViewClient

In the constructor of AppearanceService, the InitConverterThread is started, creating a new instance of ProcessToTextConverter, which is returned to AppearanceService.

Because of a implemented *appearance registration* in the proViewClient, changes for the new NLAppearance aren't necessary.

The generation of the natural language process description starts in the NLAppearance, directly after a process model is updated. The NLAppearance calls the ProcessToTextConverter's method *convertToHTML*. From this point, all operations are done in the PTTT prototype as described in Section 2.2.

3.4.2. Changes in Classes of the ProcessToTextTransformer Prototype

To match every sentence with its respective activity, an *id* attribute is implemented in the DSynT class. Because of the sentence aggregation, described in Section 2.2.3, each

DSynT can hold up to two ids, a mainId and an optionalId. The ids are given to the DSyntTs in text planning phase, at their creation.

The process description in the proViewClient shall be represented by a HTML-based string. Therefore, the class SurfaceRealizer, which realizes the sentences and combined them to the text, of the PTTT prototype has to be changed, to return a HTML-based string instead of a simple one. A new algorithm has to be created to transform DSynTs, as described in Section 2.2.3, into a HTML-based string.

The algorithm goes through a list of DSyntS, already in right order of occurrence. For each activity, the algorithm creates a paragraph tag, with the respective sentence in it. Additionally, the tag saves respective activity id in its id attribute. For each gateway, a new unordered list tag is created, representing the area between the splitting and the according joining gateway. For each branch within such an area, a list item tag is created, representing one branch from the splitting to the joining gateway. Furthermore, respective closing tags shall be created in order to keep an correct structure. To recognize new gateways, both joining and splitting, the *sen_level* attribute of DSynT's are used. A deeper level than the previous means, that there is a minimum of one new gateway, splitting the control flow. A lower level than the previous means, that there is a minimum of one joining gateway. Additionally, a attribute for bullet points is used. It is a boolean called *sen_hasBullet*. An simple example of one sentence with a tag and id looks like:

' The waiter prepares the bill. '.

The basic algorithm for sentence realization to HTML is shown in Section A.1. The *level* and *lastlevel* variables are the sen_Level of the DSynTs. First, with help of the RealPro realizer, the given XML-based DSynT is transformed into a natural language sentence. Then, multiple checks for gateways, either joining or splitting, has to be made, using the level attribute. If the level of the current sentence is 0, this means that it is in no gateway branch. Therefore, the only thing left is to close possible '<li>' and '<ul>' tags until the right level is reached. If the level is greater than the last level, a new splitting gateway is the cause. Therefore, opening according tags has to be performed. A smaller level than the previous one is caused by one or more joining gateways. As a result, according tags has to be closed. However, a check for bullet points has to be performed, cause the

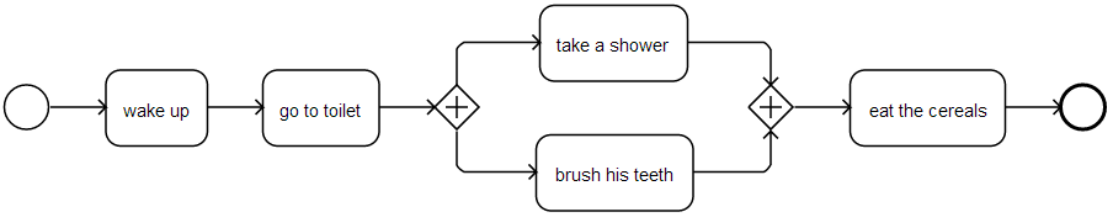

Figure 3.5.: A Students Morning Routine

current sentence might be the beginning of a new branch. The last case is the same level of the current and previous sentence. This is caused by either two sentences in the same branch or a new sentence in a new branch of the same gateway. Therefore, the sen_hasBullet attribute has to be checked. An example process, shown in Figure 3.5, is translated into following text:

```
<p>The process begins, when the student wakes up.</p>
<p> Then, he goes to the toilet.</p>
<p> Afterwards, the process is split into
                  two parallel branches: </p>
 <ul>
 <li><p> The student takes a shower. </p> </li>
 <li><p> The student brushes his teeth.</p></li>
</ul>
<p> Once all two branches were executed, the student
        eats the cereals. </p>
<p> Subsequently, the process is finished. </p>
```

The missing tags for a valid HTML document are no problem, because the NLTextArea integrates the natural langauge process description as part of the website.

The given algorithm described in the context of generating natural language texts from business process models [7] is still used for subprocess generation, where it is not necessary to have a HTML document as representation of the natural language process description.

3.4.3. The proView Prototype Template to ProcessToTextTransformer Process Structure Translation Algorithm

Remembering Section 2.1.1 and Section 2.2.5, a solution to the different process model structures has to be found. As mentioned earlier in Section 3.3, a TempToNLConverter is created to foster this issue. This section describes the algorithm for proView prototype template to PTTT process structure translation.

One problem by translating the proView template into the PTTT process structure is the different solution of agent handling. In the proView prototype, an agent is saved in each node, while in the PTTT prototype, the agents are saved in classes (*Lanes* and *Pools*).

The implementation for this translation is shown in Section A.2. First, it iterates through all nodes of the proView prototype template, generating according elements of the PTTT process structure. While creating, the agents of the activities are used for the generation and assignment of respective Pools and Lanes to the element. If a new agent occurs, a new lane is created, else the according existing one is used. Then, the algorithm iterates through all control edges of the proView prototype template, creating *Arc* elements with an unique identifier, the control edges label and their respective source and target elements. All created Elements (Activites, Arcs, and Events) are added to the ProcessModel.

However, the implementation exists in a different form for subprocesses. The only changes are, that the start and end event of a subprocess need to be added manually. This is required because of the missing start and end nodes in the subprocess description extracted from the proView prototype template.

The PTTT prototype doesn't know data elements like the proView framework. These data elements should be holding their respective id, to have a matching between text and process model. Because the natural language text is represented by a HTML document, the '' tag is chosen for representing data elements. Their id is saved in the id attribute of the tag.

To foster this issue, a template-based solution is applied in the step of surface realization. If an activity reads data elements the string ',reading [respective data elements separated

with comma]' is added to the end of the activity corresponding sentence. If an activity writes a data element, the string ',writing [respective data elements separated with comma]' is added. For the case of an activity reading and writing data elements, first the string for reading, then the string for writing is, separated with ' and ', is applied. The example in Section 3.4.2 of a HTML sentence, added with a reading of data element *customer orders* and a writing of data element *bill counter*, looks like follows:

' The waiter prepares the bill, reading customer orders and writing bill counter. '

3.5. Personalization Features

The ProcessToTextTransformer is able to personalize natural language-based process descriptions by providing a specific agent at start of the generation process [7]. For this, a connection between the proView account administration and the created instance of the PTTT prototype is made. If a logged in user's role occurs in the natural language-based process descriptions, it is replaced by 'you'.

Additionally, an export feature for natural language-based process descriptions is made, using the iText library [18]. The generated PDF file is shown in a new window. For this, a new class has to be implemented, called *PDFExporter*. To have a company header at each new page, the nested class *CompanyHeader* inherits the class *PdfPageEventHelper*. Whenever a new page is generated, a so called *NewPageEvent* is fired, triggering the *onStartPage()* method of PdfPageEventHelper. Therefore, a programmer can determine what shall happen when a new page is generated by overriding this method. Luckily, iText provides the *XMLWorkerHelper* class that automatic generates PDF documents out of valid HTML documents [18]. Therefore, no further algorithms are needed for HTML to PDF conversion. An overview over the PDFExporter class is shown in Figure 3.6.

In Vaadin, opening a resource is done with the method *open(resource: Resource, windowName: String, tryOpenAsPopup: boolean)* of class *Page* [16]. A *FileResource*, which inherits *Resource* is a file or directory on the local filesystem [16]. The *window-*

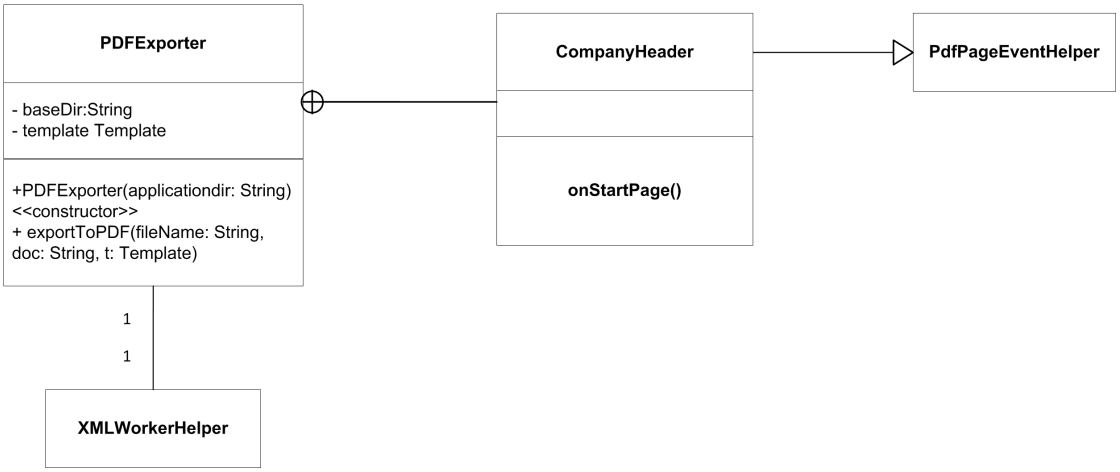

Figure 3.6.: The PDFExporter Class

Name is the name of the newly opened window. If the parameter *tryOpenAsPopup* is set true, it is tried to force the window to open in a new window instead of a new browser tab. So, to open the PDF file, one has to created a *File*, write the PDF content in it, wraps it in a FileResource, and opens this FileResource with the method mentioned before.

To create a valid HTML document, a *HTMLParser* class is implemented. It can sanitize HTML documents, i.e., wrapping the content with '<html>', '<body>' and '<head>' tags correctly. Additionally, it adds the respective doc-type. The strings for this purpose are hold in static variables called *HTML_START* and *HTML_END*. The HTMLParser can prepare HTML documents for PDF export, i.e. sanitizing the HTML document and removing all paragraph tags. This is required, because the XMLWorkerHelper integrates all line breaks of paragraph. For readers unexperienced in HTML: Before and after a paragraph, a line break performs, just how it should be in a normal texts paragraph.

However, the HTMLParser has two more methods used later on for natural language modeling. They are called *getTextFromNode(n: Node)* and *getIdFromNode(n: Node)*. Both are receiving an 'org.w3c.dom.Node' and extract the respective value. An overview over the HTMLParser class is shown in Figure 3.7.

The process of exporting a natural langauge process description is shown in Figure 3.8.

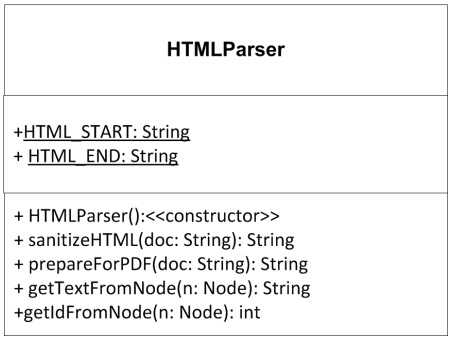

Figure 3.7.: The HTMLParser Class

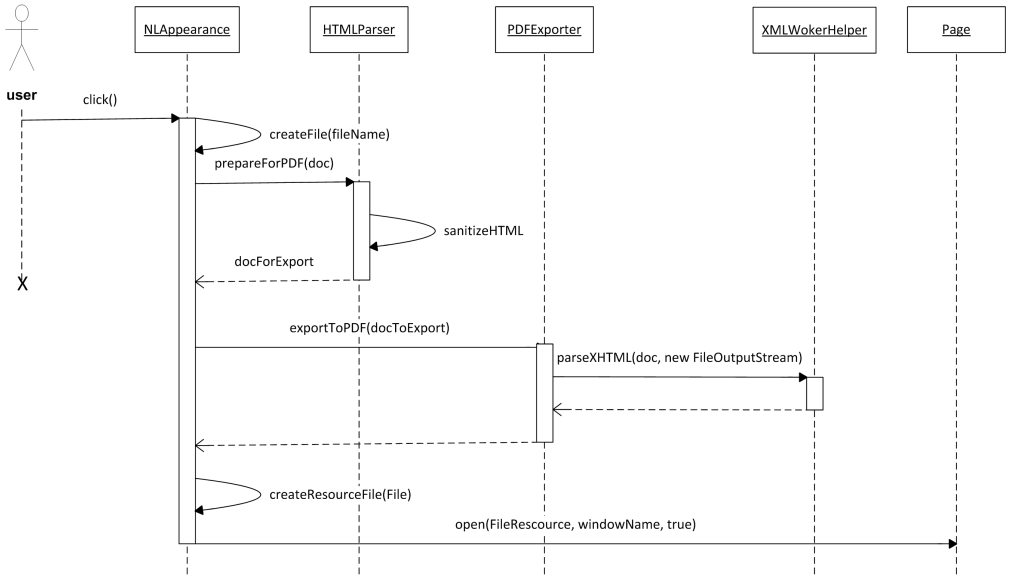

Figure 3.8.: The Process of Exporting a Natural Language Process Description as PDF

When the user clicks the *PDF Export* button, the *click()* method in NLAppearance is triggered. Then NLAppearance creates a temporary file in the file system. Afterwards, NLAppearance calls the *prepareForPDF* method from the HTMLParser. Subsequently, the HTMLParser removes all paragraph tags in the HTML-based string and calls its own *sanitizeHTML* method to create a valid HTML document string. After the HTML document is valid, it is returned to the NLAppearance. Then, the NLAppearance calls the PDFExporter's method *exportToPDF*. With the help of a call of the XMLWorkerHelper's method *parseXHTML*, the HTML document is written in the file created by NLAppearance.

After this is done, the NLAppearance creates a FileResource with the previous created file. Then, the NLAppearance calls the Page's *open* method. As result of this sequence, a new window is opened where the generated PDF is shown in the browsers native PDF viewer.

4

Natural Language-based Modeling

This Section discusses the modeling of process models based on changes in a natural language process description. Section 4.1 introduces challenges of natural language-based process modeling.

The goal after this section is, that a user can model a process in the proView project either with her mouse, or write changes directly into the natural language process descriptions that affect the process model.

Therefore, Section 4.2 introduces two approaches for natural language-based modeling.

4.1. Challenges in Natural Language-based Process Modeling

When facing natural language processing in the context of process modeling, four challenges are important [19]:

- **C1: Semantics and Syntax:** The difference between semantic and syntactic layer of a text.
- **C2: Atomicity:** Which part of a sentence should be integrated in a process model?
- **C3: Relevance:** Is a sentence relevant for the process model?
- **C4: Referencing:** How should relative references between word or sentences, and their content, be resolved?

In the following, the different challenges are described within distinct sections. Section 4.1.1 describes the challenge C1. Section 4.1.2 introduces in challenge C2. Section 4.1.3 shows challenge C3. Finally, Section 4.1.4 describes challenge C4.

4.1.1. C1: Semantics and Syntax

To describe one semantic concept, there are multiple possible syntactic patterns in a language [20]. Furthermore, semantic concepts and syntax structures are not necessarily related to each other. However, to create a process model, it is inevitable to extract the *agent*, *action* and *business object*, out of sentences. For example: an activity shall be labeled in an verb/object style. Therefore, the respective resource and action must be extracted. If one want to create Lanes (in BPMN) or, as in proView, save the agent in the activity, the agent of the sentence must be extracted as well. As an example see following two sentences:

- The waiter takes the order.
- The order is taken by the waiter.

While 'waiter' is the syntactic and semantical subject, in sentence two, the 'order' is the syntactical subject and 'waiter' is only named in a prepositional phrase. However, the semantic meaning of these two sentences is still the same.

One of the most difficult problems is the recognition of rhetoric structures [19], since conditions and orders of activites are important to process modeling. For example: 'The personal informations of new customers are asked.'

As you can see, the activity of asking for personal informations is only needed in case of a new customer. In case of a already known customer, this activity can be skipped.

4.1.2. C2: Atomicity

The challenges of Atomicity fosters the problem of which parts of a text shall be mapped to process model tasks. It is possible, that there is a 1-1 mapping for some sentences. However, a sentence like [19]: 'The GO or the MPON confirms the invoice with payment advice to the MPOO or the MSPO, or the GO or the MPON rejects the invoice of the MPOO or the MSPO.' The phrase requires four activities to be mapped correctly, i.e., (GO: 'confirm invoice', 'reject invoice') and (MPON: 'confirm invoice', 'reject invoice'). There is also the possibility of having one activity split into several sentences [19]: 'Then the food is prepared. That is done by the kitchen.' The reference done with the word 'that' indicates, that the second sentence adds further information to the first one. Therefore, a check of whether a combination of two sentences, to create one activity, makes sense.

4.1.3. C3: Relevance

Textual process descriptions, made by a process participant, can consist of examples, to clarify abstract parts of the business process [19]. However, these examples are not wanted in a process model, because it should be a generalized, abstract, representation of the describing business process.

Furthermore, the issue of using meta level descriptions, to describe the process, is important. Using meta level descriptions means, that an author doesn't explain the

process step to be conducted, but the process model. This results in phrases as follows [19]:

- 'After the Process starts, a Task is performed to [...].'
- 'If the design fails the test, then it is sent back to the first Activity.'

Since the information of these sentences will be explicitly presented in the process model, these additional meta informations are just an interference, to complicate the parsing of the text.

With the given examples, the need for an effective filtering technique, where certain relative sentences, examples and meta informations are identified and ignored, is stated.

4.1.4. C4: Referencing

Referencing can be divided into different categories. These are: *anaphoric* and *textual* [19]. Textual references consist of *forward*, *backward*, and *jump* references.

To produce process models from natural language text, anaphoric references are a problem. Anaphoras includes pronouns (e.g. 'my', 'he', 'who'), determiners (e.g. , 'this'), or phrases that describe one object with different expressions (e.g., 'Angela Merkel', 'Federal Chancellor of Germany') [19]. These Anaphoras has to be resolved for the generation step, to ensure a correct process model. Furthermore, determiners like 'there' are a special problem. For example: The sentence 'There are times, when [...]' refers to a concept which isn't linguistically described in the text, but in its context. The word 'it' takes a special place as well, since it can be used as a pronoun, but also as an emphasizing word (e.g., 'Sometimes it also happens [...]'[19]).

A forwards reference is used when the author wants to describe an alternative path to the same goal as in the main control flow. For Example [19]: 'Of course, asking the customer whether **he is generally interested** is also important. If this is not the case, we leave him alone, except if the potential project budget is huge. Then the head of development personally tries to acquire the customer. **If the customer is interested in the end**, the next step is [...]'

In the first sentence, the customer can be interested or not. Thus, two branches are generated. The following describes the goal of having getting interested customer. So, there is one branch, that is executed when having a interested customer, and one, that describes a way to get an uninterested customer interested. Therefore, if the second branch can get the customers interest, the control flow can go on as if it was using the first branch.

A backwards reference is used for loops, whenever a task is done repetitively. Example sentences could be:

- 'The proposal is checked again.'
- 'The proposal is send back.'
- 'The next proposal is checked'

All these sentence use indicating words for loops, i.e., 'again', 'back', and 'next'. Therefore, an algorithm may use this fact for loop detection.

Jumps are used in processes with different results [19]. In contrary to forward references, an author follows one branch after a split and then returns to the split position for the next branches description. For example: 'The customer can pay with credit card or cash. If the customer pays with credit card, he [...]. If the customer pay cash, he [...].'

All three types of textual references have to be recognized in process model generation. Problems are whenever meta information is used to describe the reference. In those cases, the syntactic level can't resolve the reference, and it has to be done on semantic level. This requires knowledge of the domain [19]. Implicit conditions are another problem that's resolution need domain knowledge, sine implicit conditions don't exist on syntactic level.

4.2. Approaches to Natural Language-based Process Modeling

This section introduces two approaches in the proView project natural language-based process modeling. Section 4.2.1 introduces a mouse-based modeling approach. Section 4.2.2 discusses a text-based approach.

4.2.1. A Mouse-based Natural Language Process Modeling Approach in the proView Project

In the proView prototype, a mouse-based process modeling approach is used for process modeling. There, a process modeler can select nodes and perform operations proposed by the proView prototype according to the selection. Figure 4.1 shows a selected node (1) and the proposed create and update operations (2). The proView prototype now combines this modeling mechanics with the natural language process description. A process modeler can select the activities within the text, and on right click, the proView prototype proposes operations.

Figure 4.2 shows the same process of Figure 4.1 in natural language process description with two selected sentences (a) and the possible operations (b). Furthermore, the branch and gateway highlighting mechanism is shown. When the mouse hovers a branch, the respective branch is highlighted with a dotted square (c). When it hovers an area between two corresponding gateways, this area is surrounded with an solid square (d). Furthermore, the highlighting mechanism also highlights selected sentences (a) by giving it a gray background. What can't be seen is the highlighting of data elements in the natural language process description. Whenever a data elements representation is selected, the background changes to green, when the data element is read, or red, when the data element is written. Long texts can become confusing to the reader. Therefore, this mechanism is made for better structural understanding and for increased overview of the natural language process description. With the same mechanics as in graph-based process modeling, all the operations of graph-based process modeling in the proView

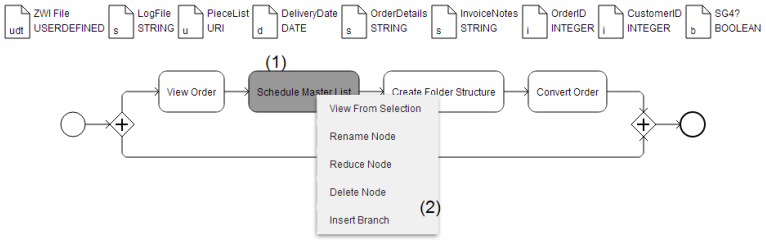

Figure 4.1.: Classic Mouse-based Process Modeling in proView

prototype are possible in the natural language text mouse-based approach. The 'add activity' operation, as example, is done as follows:

1. Select sentences as shown in Figure 4.2 and click right.
2. When right clicked, a context menu opens with the context item 'insert Node between'.
3. Select the mentioned item. A new window, shown in Figure 4.3 opens, where the user writes the new activities name.
4. Click 'OK'.
5. The new activity is created, as described in Section 2.1.2.
6. Then, the natural language process description is newly generated as described in Section 2.2.
7. Finally, the updated process model is shown as natural language process description.

C1, described in Section 4.1, has little importance, since the user doesn't use full sentences to create an activity, but labels them like in a graph-based solution. Therefore, the structural problems occurring with C1 are not possible. C2 doesn't concern in this approach as well, since the process modeling mechanism forces a 1-1 mapping of sentences. However, C3 is important. A user can add new activities to the process model, so she can add unwanted activities as well. C4 is also of great concern, since referencing in process models shouldn't be done. An approach to cut the use of references, in this context especially Anaphoras, is the spacial differentiation between process text and the

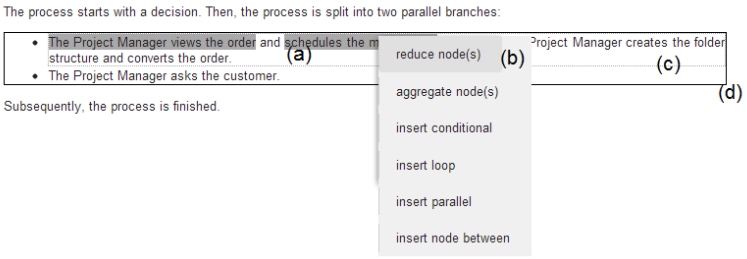

Figure 4.2.: Natural Language Mouse-based Process Modeling in proView

input interface. Whenever an activity shall be added or updated, a new window opens where the label for the activity can be written. The user can't see the natural language process description and, hopefully, may not be ensnared to use references.

The semantical level still needs to be tackled by the user. However, since the structure of a process model is solely changed with given operations, the structural problems, concerning natural language process modeling, are solved with this approach.

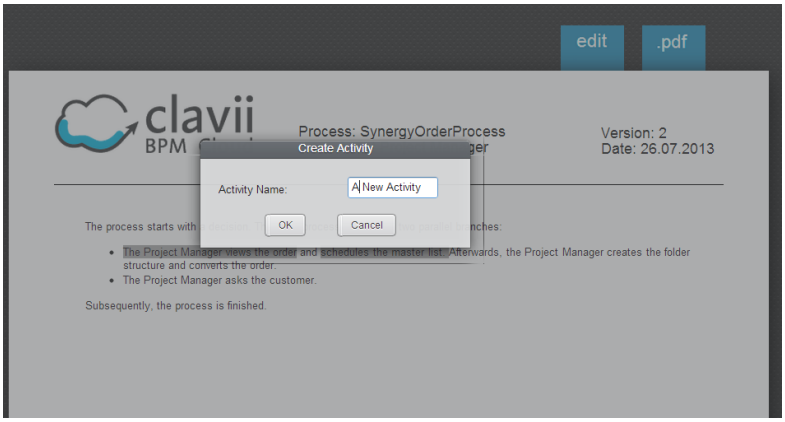

Figure 4.3.: Add Activity Window

4.2.2. A Text-based Natural Language Process Modeling Approach in the proView Project

In contrary to the mouse-based process modeling approach described in Section 4.2.1, the text-based process modeling approach allows the user to freely write in the generated

natural language process description. Then, after the user has saved his changes, the differences between the original text and the user changed one are resolved. The found differences lead to the use of operations mentioned in Section 2.1.1 until all differences are integrated in the process model. This approach divides the structural and semantical level of processes, i.e. the structure is given by the texts structure and the semantic is given by each sentence. Therefore, all structure changing operations of proView have to be mapped on text structures, shown in Table 4.1.

Operation	Change in Text
Delete Activity	Delete corresponding sentence
Add Activity	Write new sentence
Add Gateway	Indicating sentence and new bullet point
Add Data Element	Sentence context (element not used in text until now)
Delete Data Element	Sentence context (element not used in text)
Reduce Activity(-ies)	Set sentences in brackets
Aggregate Activites	Set sentences in brackets and write a replacing sentence in front
Create View From Selection	(-)
Show Subprocess	When hovering a sentence a tool-tip pops up
Reduce Data Element(s)	(-)
Aggregate Data Element(s)	(-)

Table 4.1.: View Update Operations

The add and delete operations for activities are straight forward: a new sentence is a new activity, a missing sentence is deletion of an activity. However, adding a gateway is a bit more complex. When the writer wants to add a new gateway, she first writes an introducing sentence, where the type of the gateway must be explained (e.g., 'AND', 'XOR', 'LOOP'), this can happen either with meta information, or a sentence, that semantic is clearly targeting one type. This sentence needs to be ended with a colon, indicating that a new gateway shall be created. When pressing enter after a colon, a new bullet point is created. There, the user can write new sentences to fill the branch

with activities. So, to recognize the textual structures at writing, proView listens to key input and identifies the input of colons, enter combination.

Data elements are a special treat in this approach. Because data elements are only represented in the sentences of activities, they can only be modeled by using the sentence context. Adding a new data element means, that a sentence is writing, because data has to be written before it can be read, a new data element, unknown to the process model. Deleting a data element means to delete all occurrences of the respective data element in every sentence. Therefore, the adding and deleting of data elements is done in the semantical part.

For the create view operations ('reduce', 'aggregate'), the writer has to set the sentences to be reduced or aggregates in brackets. The difference between the respective operations is, that for aggregation, a sentence that replaces the elements to be aggregated must be inserted before the brackets. An Example:

- **Reduce:** (The waiter takes the order. Then, he serves the food.)
- **Aggregation:** The waiter serves the customer (The waiter takes the order. Then, he serves the food.).

As one can see, the brackets at the aggregation stand within the replacing sentence. The new abstract activity would be *'Wait on Customer'*.

However, the reduction and aggregation of data elements, because of their multiple occurrence, is not possible in this approach. Consider the following: The writer reduces data elements in one sentence, but they occur in another one. There a two possible outcomes from this action: First, the new abstract data element is only used by this activity. Second, all activities using these data elements are writing and reading now in the abstract data element.

Finally, a new view from selected elements cannot be performed, because in this approach, there is no selection.

On semantical level, an algorithm extracts Agent, Action, and Resource of each sentence. Furthermore, it recognizes data elements and calls according operations.

In order to have this approach work, some rules for the writer have to be made:

- One sentence is one activity.
- Don't reference.
- Don't writes useless sentences.

With these constraints and the division of structural and semantic level, the challenges C1-4 are weakened enough to have this approach work. With no references, there can be no structural information in sentences and there is a 1-1 mapping of sentences and elements forced. The third constraint is not necessarily needed for this approach, because this is a general constraint for all process modelers.

So, after the writer writes his changes in Step 1 and saved them, Step 2 is processing. There, the differences between the old process view and the new process description are identified. Then, according to the differences, the create and update operations are called. First, all ids, that are no more in the natural language description are delete in the process view. Then the process views elements and the process descriptions sentence are iterated from start to end. In every iteration, the ids of the current sentence and element are compared. If they match, the semantic information is compared (agent, action, and resource). If they match, the next iteration step starts, else an update operation is performed. If the ids don't match, the sentence is evaluated, whether it is an activity or gateway. If it is an activity, a new activity is added. If it is an gateway, a new splitting gateway with the all its branches and their activities, and the corresponding joining gateway are added.

Because we have made the constraints before, we can easily determine the actors, actions and resources with the help of the Stanford parser. First, we tag the sentence with the Stanford parser. Then, we determine the voice(active or passive). This can be accomplished by searching for the occurrence of a present form of 'be' and a past participle verb. If the sentence is in active voice, then all nouns before the first verb are the agent, the verb is the action and all nouns behind are the resource. In passive voice, the past participle verb is the action, and the previous nouns are the resource. In passive voice, we have to further determine if there is a prepositional phrase that describes the agent. Therefore, we check for prepositional phrase indicators e.g. 'by'. The noun in the prepositional phrase is used as agent. For example, consider the sentences:

- The waiter takes the order.
- The order is taken by the waiter.

Their respective tagged sentences are:

- The(DT) waiter(NN) takes(VBZ) the(DT) order(NN).
- The(DT) order(NN) is(VBZ) taken(VBN) by
  IN the(DT) waiter(NN).

'(DT)' stands for a determiner, '(NN)' stands for noun (singular or mass), '(VBZ)' stands for verb, third person, singular, present, and '(VBN)' stands for verb, past participle. [11]

Because the first sentence is active voice, we take 'waiter' as agent, 'takes', in base form, as action, and 'order' as resource. Because the second sentence is passive, we take 'order' as resource and 'taken', in base form, as action. Furthermore, the preposition phrase indicator 'by' appears, so we take 'waiter' as agent. For both sentences, the new activity would be labeled 'Take Order' with the agent 'waiter'.

As example for process modeling, a process view and three changes are shown in Figures 4.4, 4.5, 4.6, and 4.7. The changes made in the natural language process description are as follows:

First, the natural language generated text from the start process view:

```
1 The process starts with a decision. Then, the process is split
2 into two parallel branches:
3   -The Project Manager views the order and schedules the
4    master list. Afterwards, the Project Manager creates the
5    folder structure and converts the order.
6
7   -The Project Manager asks the customer.
8 Subsequently, the process is finished.
```

In the first step, we recognize that it would be appropriate for the project manager to evaluate the answers of the customer on his questions. Therefore, we add an according sentence to the process description:

```
The process starts with a decision. Then, the process is split
into two parallel branches:
  -The Project Manager views the order and schedules the
   master list. Afterwards, the Project Manager creates the
   folder structure and converts the order.

  -The Project Manager asks the customer. Then,
   the Project Manager evaluates the customer answers.
Subsequently, the process is finished.
```

However, we miss the project managers duty of budget planning. So we add a new bullet point and sentence to the process description:

```
The process starts with a decision. Then, the process is split
into two parallel branches:
  - The Project Manager views the order and schedules the
    master list. Afterwards, the Project Manager creates the
    folder structure and converts the order.

  - The Project Manager asks the customer. Then,
    the Project Manager evaluates the customer answers.

  - The Project Manager manages the project budget.
Subsequently, the process is finished.
```

Well, after another look on the process view, we decide that we don't need the trivial information of asking a customer and then evaluating the answers. Therefore, we want to combine those two sentences. Because the PTTT prototype uses the word 'and', as described in Section 2.2.3. We have to use an more nonnative way to explain what we want:

```
The process starts with a decision. Then, the process is split
into two parallel branches:
```

```
  - The Project Manager views the order and schedules the
    master list. Afterwards, the Project Manager creates the
    folder structure and converts the order.

  - The Project Manager handles the customer(The Project
    Manager asks the customer. Then, the Project Manager
    evaluates the customer answers.).

  - The Project Manager manages the project budget.
Subsequently, the process is finished.
```

By bracketing the sentences that shall be combined and setting a new sentence in front, we tell proView to use the *Aggregate(ID1, ID2)* operation. Therefore, proView needs to identify such constructs correctly. Then, it can use the ids saved within the sentences, remember Section 3.4.2, to aggregate the respective activities. The name for the abstract activity is extracted from the sentence before the brackets.

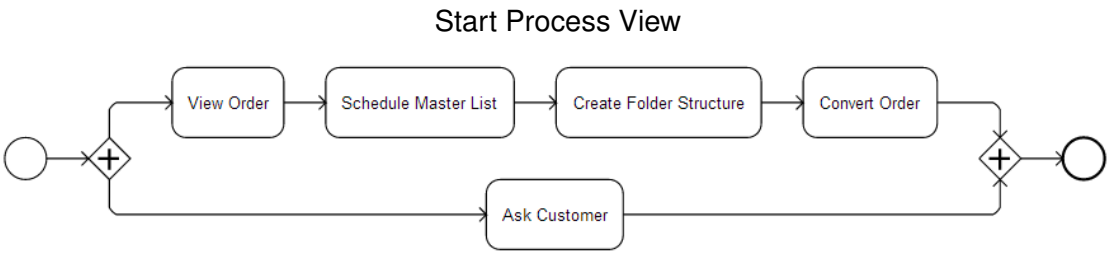

Figure 4.4.: Example Process Modeling: Basic

Add_Activity('Evaluate Customer Answers')

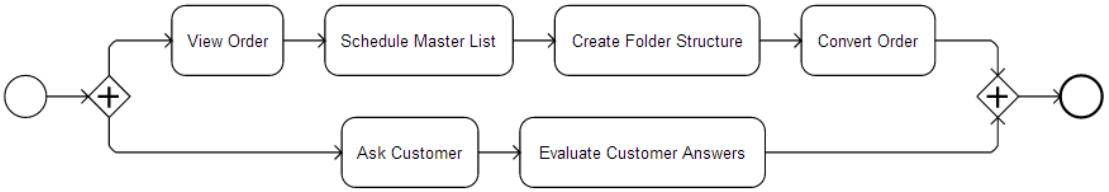

Figure 4.5.: Example Process Modeling: Change one

Add_Activity_Parallel('Manage Project Budget')

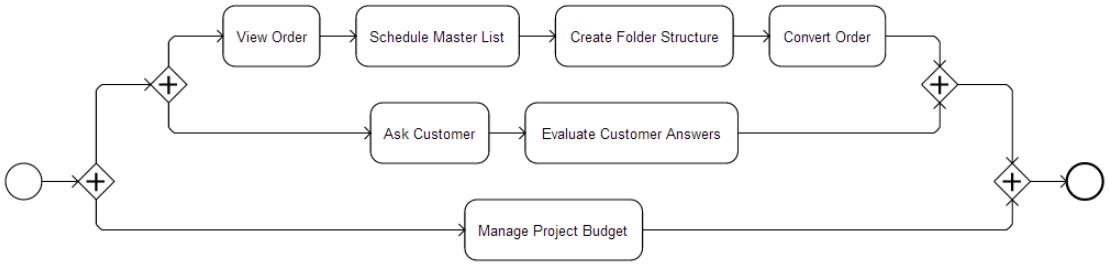

Figure 4.6.: Example Process Modeling: Change two

Aggregate('Ask Customer', 'Evaluate Customer Answers')

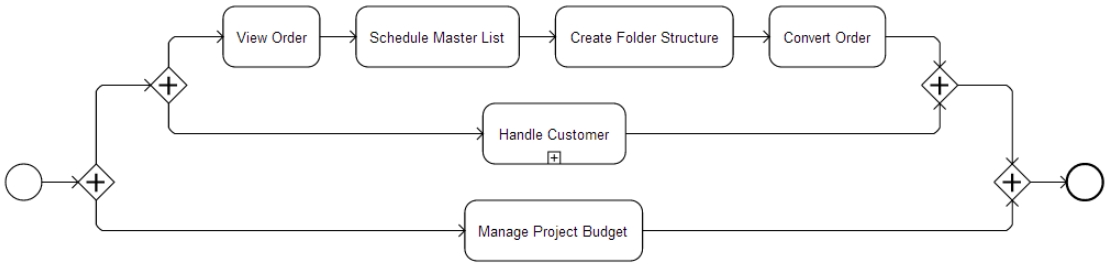

Figure 4.7.: Example Process Modeling: Change three

5

Implementation of Natural Language Process Modeling Approaches in the proView Prototype

This chapter introduces the implementation of the mouse-based and text-based process modeling approach described in Section 4.2.1 and 4.2.2. Section 5.1 introduces the NLTextArea component. Section 5.2 shows the implementation of the mouse-based process modeling approach. Section 5.3 introduces the implementation of the text-based process modeling approach.

5.1. The Natural Language Text Area

As mentioned in Section 3.4.2, the natural language process description is structured as a HTML-based text, using '<p>' for activities, '<ul>' for gateways, and '<li>' for branches. The mouse-based approach needs to detect click events on activities, for modeling mechanics. Therefore, the *Natural Language Text Area (NLTextArea)* is created.

Generally, when integrating a Java script library in Vaadin, there are four components that represent the whole component. The *client-side widget* is the Java script library to be integrated. The *connector* connects the client-side widget with the server. The *server-side component* is a Java class that represents the Java script library on the server. Finally, the *component state* is a shared state object between the server and client side [21]. The server-client communication is done over an *UIDL* stream, where the values are serialized in a Json array, sent, and then are deserialized again [1]. The communication over the UIDL stream can be performed on two ways: For static messages such as attributes, the respective value can be saved in the component state where each side has access. For dynamic issues, both sides can use remote procedure calls on Java script functions to communicate with the respective other side. To call a server-side function from the client, the respective component needs to register the function first to the connector, using the AbstractJavaScriptComponent's *addFuntion* method. This method has two parameter: the name of the function and a JavaScriptFunction class. This JavaScriptFuntion class has a method called *call*, which gets called when the respective Java script function is called. When the client-side wants to call a function, the connector simply calls it by using the point operator: 'connector.theFunction()'. To call a Java script function of the client-side, the AbstractJavaScriptComponent provides the method *callFunction(name, values)*. The server-side component calls this method with the respective Java script functions name and values to transmit. Only functions in the connector can be called this way.

An overview over the NLTextArea structure is shown in Figure 5.1. The *NLTextAreaWidget* is the client-side library. The *NLTextAreaConnector* is the connector library. The *NLTextAreaState* is the shared state class. The *NLTextAreaComponent* is the server-side component.

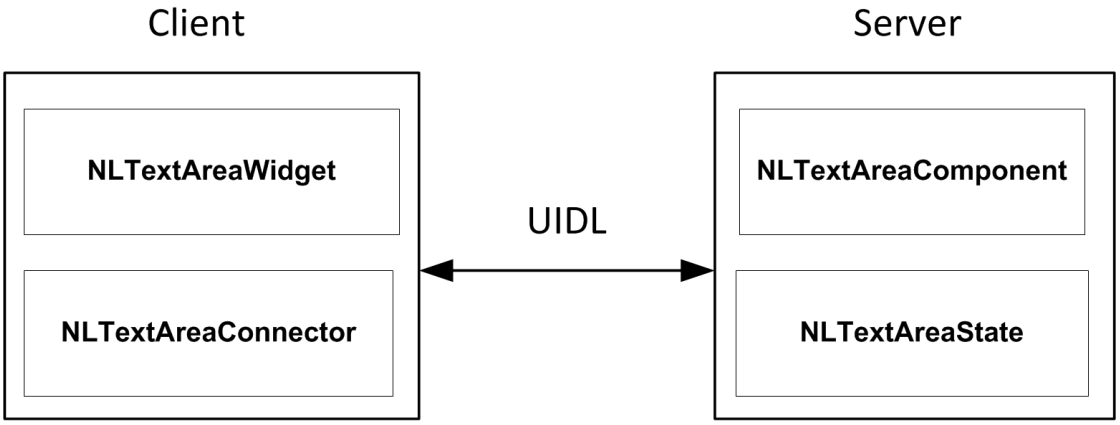

Figure 5.1.: Overview over the NLTextArea

The server-side classes are shown in figure 5.2. Both the component and the state are inheriting a respective Vaadin class. This is done, so Vaadin can take care of the fundamentals behind UIDL streaming to the client. Because the component detects clicks on client side, a clickOption attribute saves whether the click was performed with left or right mouse button, or if the ctrl key was pressed while clicking. Therefore, static finalized attributes for each possibility are created. An listener interface for each clicks on activity and data elements are created. A class that wants to know if a click was performed need to implement the interface and register itself with the according add method. To receive the id of the clicked element, the getId method is used. Because the component shall be able to either detect mouse clicks for the mouse-based process modeling approach or edit text for the text-based process modeling approach, the setClickEdit function can be used to switch between these two modes. Furthermore, to permit editing or not, the AbstractJavaScriptComponent class provides an according setReadOnly function.

The state class holds the HTML-based process description as string and also the id of the clicked element. Furthermore, it holds the boolean for edit mode decision. It has according getters and setters for the attributes. The NLTextAreaComponent has three added Java script functions: *onTaskClick, onDataElementClick, and update*. The first two are used for receiving the id of elements that are clicked in the HTML-based text. Therefore, they save the id and the type of click in the component state and then

inform all registered listeners. The update function is used to get the current text in the client-side widget in the component state.

The source code for the client-side widget is shown in Section A.3. It consists of following functions:

- **setID/getID:** Getter and setter for the id of the clicked element.
- **setClickMode:** Function for swtiching between click and edit mode.
- **getText/setText:** Getter and setter for the natural language process description.
- **setReadOnly:** Function that changes the permission for editing.
- **click:** Function that is called when an element is clicked.
- **colorSelectedTasks/colorSelectedDataElements:** Functions for element highlighting mechanism.

The click function is implemented in the connector library. It receives a string parameter that describes if the detected click in the div container was done with the left or right mouse button, or with the left mouse button while pressed the ctrl key. Then, the function calls via remote procedure call the Java script functions added to the server-side component.

The client-side widget has a div container that holds the natural language process description. Therefore, the setReadOnly function changes the divs attributes onclick, oncontextmenu, and contentEditable according to the current edit mode and the received boolean. When in clickEdit mode, a function is set in the onclick and oncontextmenu attribute, that check if the clicked element was a paragraph or span tag and then calls the components click function with according parameters.

The functions colorSelectedTasks and colorSelectedDataElements, used for the highlighting mechanism described in Section 4.2.1, receive an integer array with the ids of elements that are to mark. Therefore, the respective function iterates through the list of elements received with 'getElementsByTags' and marks the according elements by setting the style.background attribute. If an id is not in the list to mark, the background is set null, to remove possible marks from a previous selection.

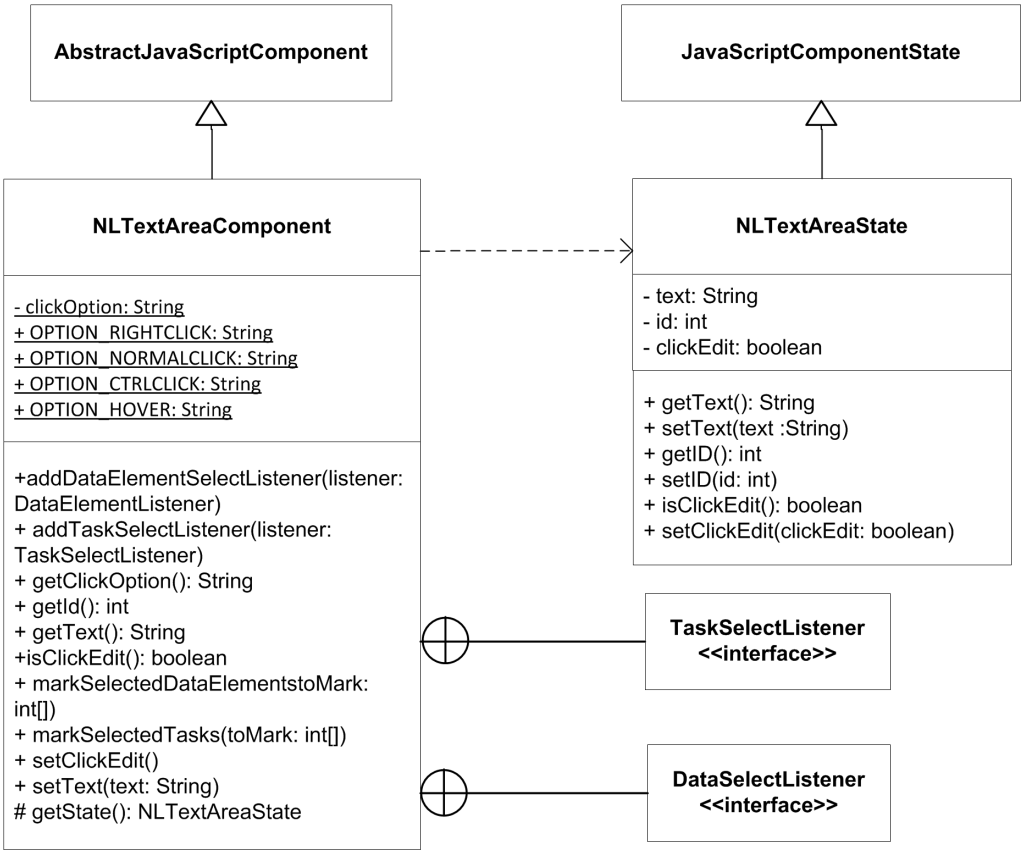

Figure 5.2.: Server-Side Classes of NLTextArea

To explain the client to server communication furthermore, consider following example, shown in Figure 5.3: The user left clicks on an activity, therefore, NLTextAreaWidgets click("left") function is called. inside this function, the NLtextAreaConnectors onTaskClick(id, "left") function with the id of the respective activity and the click operation "left" is called. This causes the call of the method *call* of the registered JavaScriptFunction with an Json array consistent of the id and "left". This method now sets the id and clickOption attribute in the shared state object. Afterwards, the method informs all registered TaskSelectListener by calling the onTaskSelect method. Now that the listener is informed, it can get the id and clickOption attribute of the shared state, by calling the respective getter of the component. The component gets the attributes of the state and

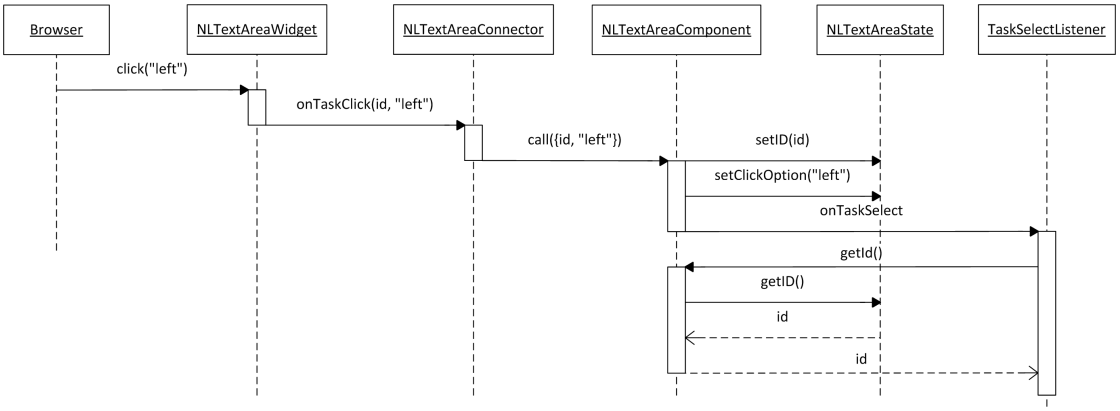

Figure 5.3.: Client to Server Communication Example

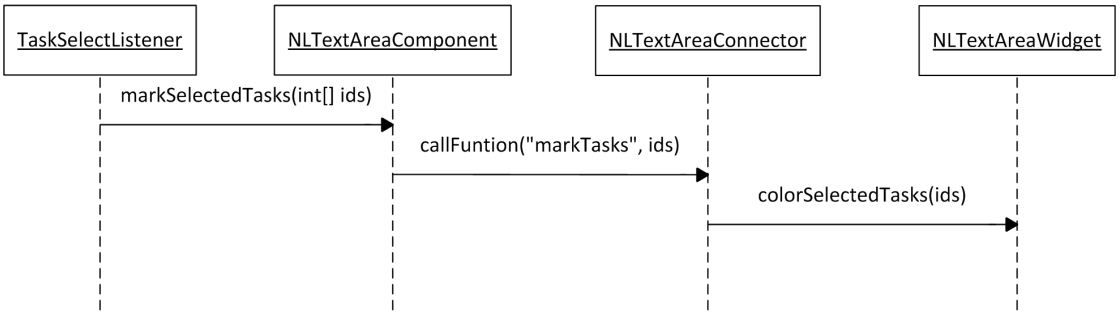

Figure 5.4.: Server to Client Communication Example

returns them to the listener. The diagram only shows getting the id attribute, but getting the clickOption works the same way.

The server to client communication is described in the following example: A user clicked an activity like in the previous example. Then, a listener may want to mark the clicked activity. The execution is shown in Figure 5.4:

The TaskSelectListener calls the NLTextAreaComponents *markSelectedTasks* method with an integer array as parameter. Then, the NLTextAreaComponent makes use of the AbstractJavaScriptComponents *callFunction* method by adding the NLTextAreaConnectors function name ('markTasks') and the integer array. Now, Vaadin automatically transforms the Java type integer array into a Json integer array and makes the call. Finally, the NLTextAreaConnector calls the NLTextAreaWidgets function *colorSelectedTasks*.

5.2. Implementation of the Mouse-based Natural Language Process Modeling Approach

To implement the approach described in Section 4.2.1, the Java script component of previous section and a new class are used. The new class is called *SelectionHandler*. It uses the *Context Menu Addon* made by Peter Lehto [22]. Therefore, SelectionHandler implements the *ContextMenuItemClickListener* interface that triggers on every click on a *ContextMenuItem*. ContextMenuItems are gathered within one *ContextMenu*. To detect clicks in the NLTextArea, the SelectionHandler is registered as TaskSelectListener and DataElementSelectListener. It holds the selected ids of activities and data elements in separated *ArrayList*s of type *Integer*. To have access on the process modeling operations of proView, the SelectionHandler holds the *ModellingService*. Furthermore, it saves the instance of the owning NLAppearance, to make use of the PTTT protoype for subprocess generation. An overview over the SelectionHandler class is shown in Figure 5.5.

Now, whenever the NLTextArea permits editing the natural language process description and is in clickEdit mode, the id is sent to the SelectionHandler as described in Section 5.1. Then, the SelectionHandler saves the id in the respective ArrayList according to the sent clickOption and the triggered Listener. If it was a left click, the ArrayList is emptied and the new id saved. If it was a right click, or a left click while pressing the ctrl button, the id is added to the respective ArrayList. Furthermore, on right click, the context menu is generated and opened. On every triggered listener, the NLTextAreaComponents markSelectedTask or markSelectedDataElements method is called, as described in Section 5.1. The possible context menu items, and therefore the operations implemented, are listed in Table 5.1.

Some operations are not yet implemented: *insertion of a loop* and *the creation of a process view of selected elements* as well as all of the data elements operations.

If a context menu item is selected, the implemented ContextMenuItemClickListener listener is triggered and calls the according operations from the ModellingService. For example: To add a new activity, the user clicks the context menu item 'Insert Sentence Between'. The respective operation of ModellingService is: 'modellingSer-

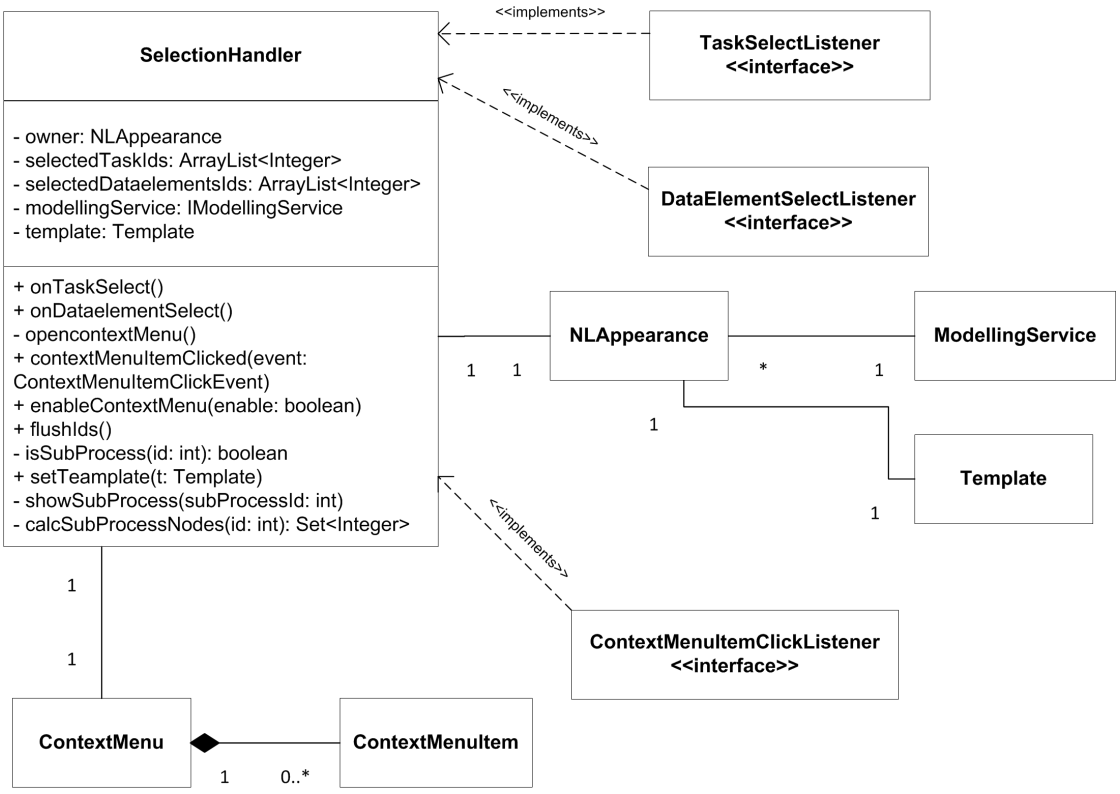

Figure 5.5.: SelectionHandler Class Diagram

Menu Item	Operation
Reduce Sentence(s)	Reduce Selected Elements
Rename Sentence	Relabel Selected Activity
Delete Sentence(s)	Delete Selected Elements
Insert After	Add Activity after Selected Activity
Aggregate Sentences	Aggregate Selected Activities
Insert Parallel	Insert new Activity and AND Gateway, Surrounding Selected Activites
Insert Conditional	Insert new Activity and XOR Gateway, Surrounding Selected Activites
Insert Sentence Between	Insert Activity Between two Selected Activites
Show Subprocess	Show Process Hidden in Abstract Activity or Subprocess

Table 5.1.: Implemented Operations

vice.insertSerial(viewId, selectednodes);' where the id of the respective view and a list of node ids as parameters are given. From this point, all is handled by the ModellingService: Create a new window to add the new name, send the operation to the server, receive the new process view and apply it to the respective appearance.

The possible context items for a given selection decides over the number of selected elements. If one element is selected, the user can decide between: rename, reduce, delete, insert after, and, if the selected element is a subprocess, show subprocess. If two elements are selected, the user can: reduce, aggregate, insert parallel, conditional, and serial. Else, i.e., more than two elements, the user can reduce and aggregate.

5.3. Implementation of the Text-based Natural Language Process Modeling Approach

For the issue of text-based process modeling, a new class is created in the proView prototype: the *TextInvestigator*. It makes use of parts of the PTTT prototype. The class *EnglishLabelHelper* provides methods to find the basic forms of words, e.g.. the method *getInfinitiveOfAction(String verb)* returns the basic form of a given verb(e.g., 'creates' returns 'create'). Furthermore, this class can tag sentences with the Stanford parser. For the text-based process modeling approach, this class is mainly used to recreate the basic form of verbs from sentences or tag them. The TextInvestigator also uses the Java library *JTidy* [23], a HTML syntax checker and DOM parser, that means JTidy can create a Java-based DOM structure of HTML documents, consistent of nodes. The HTMLParser provides methods to retrieve ids and texts from nodes created by JTidy. To be able to call the proView prototype operations, this class also has access to the ModellingService.

The TextInvestigator has six operations: *checkDeletes*, *checkUpdates*, *extractActionFromTaggedSentence*, *extractResourceFromTaggedSentence*, *extractVerbAndNounFromLabel*, and *isPassiveVoice*. The first operation checks for deleted sentences by comparing the existent ids in the old process model and the ids in the current natural language description. The missing ids must have been deleted. The second operation checks

for changes in still existing sentences. Therefore it extracts the action and resource of the sentence and label, using the respective operations three, four, and five, and compares them. If the respective resources or actions have changed, a rename is performed. The last operation determines, if a sentence is in passive or active voice. An overview over the described classes, with their important operations and attributes for this approach, is shown in Figure 5.6. The source codes for operations three, four, and six, with explanatory comments are shown in Section A.4.

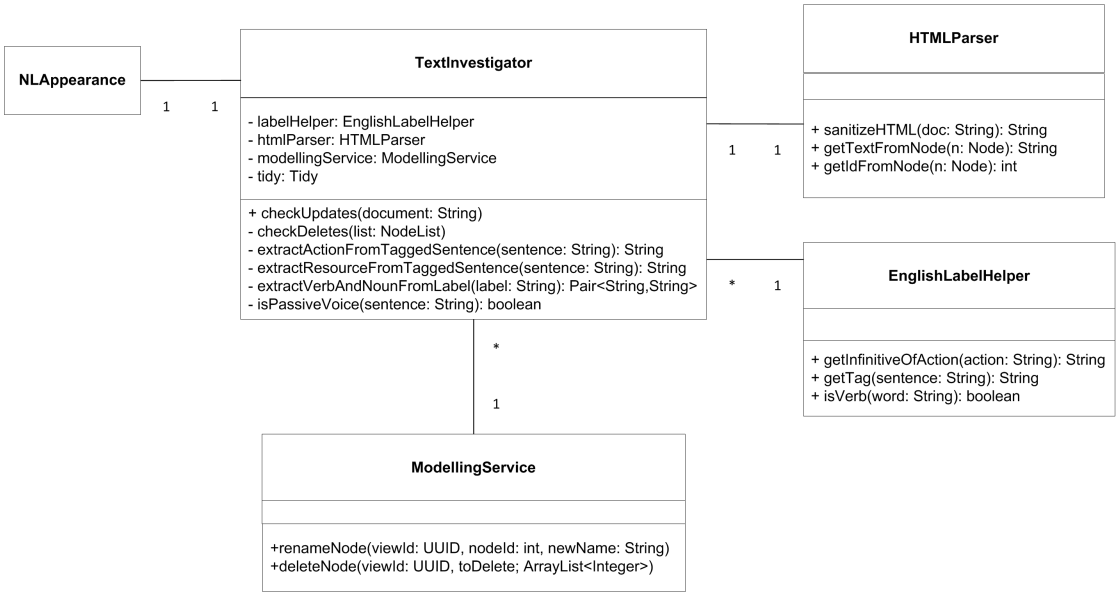

Figure 5.6.: Classes Used in Text-based Process Modeling Approach Implementation

As said in Section 5.1, the NLTextArea provides the possibility to write freely in the natural language process description. When the user ends the edit mode of the NLTextArea, by pressing the 'edit button' again, NLAppearance calls the checkUpdate method of TextInvestigator. There, the HTMLParser first sanitizes the string to a valid HTML document. Afterwards, JTidy parses a Java-based DOM structure. Subsequently, All paragraph tags in the document are saved in a list. Now, the checkDeletes method compares the ids in the document with the ids from the old process model. Therefore, it creates a list of all ids of the old process model and deletes all ids of the document in this new created list. The ids still in the new list must have been deleted. Therefore, the

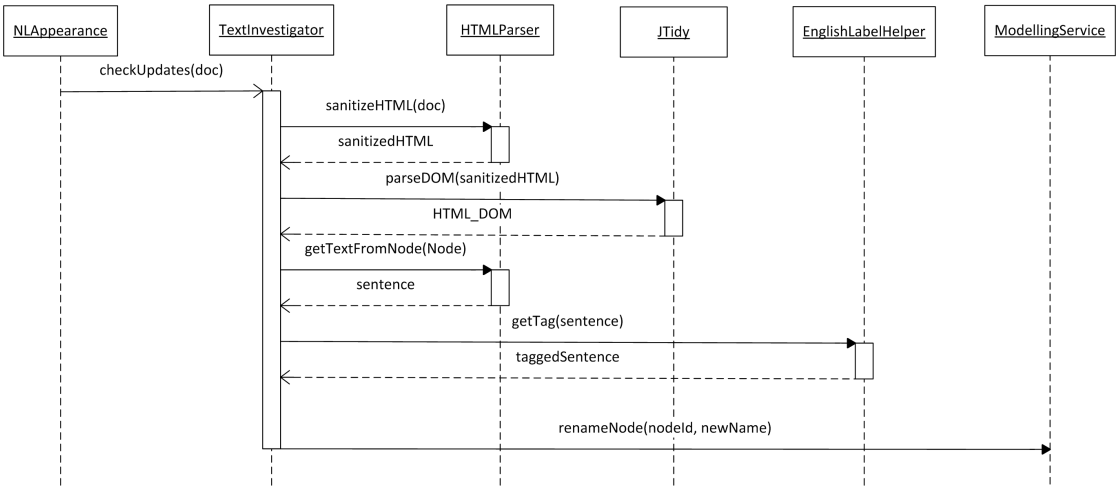

Figure 5.7.: Sample Execution of an Activity Rename

method calls the deleteNode method of ModellingService. Afterwards, the checkUpdate method iterates through the list of sentences, extract the respective action and resource, and compares them to the according action and resource of the respective label. If they match, nothing is done. Else, the method renameNode of ModellingService is called, changing the label of respective activity. An overview of the communication is given in Figure 5.7.

At the time this thesis is written, the proView prototype's text-based natural language process modeling implementation can only delete elements and rename activities.

6

Conclusion and Further Steps

In this thesis, we integrated a natural language component into the proView prototype. proView now is able to generate natural language process descriptions from its process models and provide a mouse-based process modeling solution on the generated text. Furthermore, simple update and delete operations can be performed by freely writing in the natural language process description.

The next steps are the complete integration of the text-based natural language modeling approach by further development of the NLTextArea and TextInvestigator. Then, by refining the natural language processing algorithms, a way to import text descriptions of processes and automatically generate them can be developed. Having this feature, applying a voice to process model solution by using a voice to text and then the text to process model solution is possible. Additionally to further development, the advantages of natural language process modeling have to be proofed by studies.

A

Source Codes

In this appendix, there are several important sources:

A.1. The Basic DSynT to HTML Algorithm

```
1 /*
2 * sentence is the DSynT to realize into HTML
3 * level is the level of the current DSynT
4 * lastlevel is the level of the previous DSynT
5 * This method is called by another method that iterates
6 * through a list of DSynTs.
7 */
8 realizeSentence(DSynT sentence, int level, int lastlevel){
```

```
String output = RealPro.realize(setnence);

String tags = "";
// Check for joining or splitting gateways using
// level attribute
If(level == 0){ // current sentence is in no gateway branch
 For lastlevel To level-1{
 tags += '</li></ul>';
 }
}
Else If(level > lastlevel){ // new splitting gateway(s)
 For lastlevel To level-1{
 tags += '<li><ul>';
  }
}
Else If(level < lastlevel){ // new joining gateway(s)
 For level To lastlevel-1{
 tags += '</li></ul>';
 }
 If(sentence.hasBullet){
 tags += '<li>';
 }
}
Else If(level == lastlevel){
 If(sentence.sen_hasBullet){
  tags += '</li><li>';
 }
}
If(sentence.optionalId != null){
 String[] activities = outputsplit(' and ');
 return tags + <p id="'+sentence.mainId+'">'+activites[0]+
```

```
40      '</p><p id="'+sentence.optionalId+'">'
41          +activites[1]+'</p>';
42   }
43   Else{
44    return tags + <p id="'+sentence.mainId+'">'+output+
45          '</p>';
46   }
47 }
```

A.2. The proView Template to PTTT Process Structure Algorithm

```
1
2  public ProcessModel createFromTemplate(Template template){
3    // Save all pools and lanes
4    pools = new ArrayList<Pool>();
5    lanes = new ArrayList<Lane>();
6    // Save all elements in HashMap for Arc creation
7    tasksNgates = new HashMap<Integer,Element>();
8
9    // Get all Nodes of the proView Process Model
10   Set<Node> nodes = template.getNodes();
11   // Get all Edges of the proView Process Model of type
12   // control and loop
13   Set<StructuredEdge> edges = template.getEdgeStructure(
14                                EdgeType.ET_CONTROL);
15   edges.addAll(template.getEdgeStructure(EdgeType.ET_LOOP));
16   ProcessModel p;
17   /*
18    * Filling order:
```

```
19      * 1. activites/gateways
20      * 2. arcs
21      */
22      // If Template is CPM, used template ids hashcode
23      // (id is type UUID) and name for ProcessModel
24      // id and name.
25      // new ProcessModel(id, name)
26      if(ADEPTUtils.isCPM(template))
27        p = new ProcessModel(template.getID().hashCode(),
28            template.getName());
29      // Else use process view name and ids hashcode
30      else
31        p = new ProcessModel(
32            ADEPTUtils.getViewId(template).hashCode(),
33            ADEPTUtils.getViewName(template));
34      Element temp = null;
35      // Iterate all nodes of the template
36      for(Iterator<Node> i = nodes.iterator(); i.hasNext();){
37        Node n = i.next();
38        // Get NodeType of respective Node
39        NodeType type = template.getNodeType(n.getID());
40        // Create Elements, ignore lanes and pools.
41        // This is done afterwards
42        switch(type){
43          case NT_NORMAL:
44          // Normal Activity => create new Activity
45          // new ACtivity(id, name, lane, pool, type)
46            temp = new Activity(n.getID(),
47                                n.getName(), null, null,
48                                ActivityType.NONE);
49            break;
```

```
50        case NT_AND_JOIN:
51        // Joining AND Gateway => create Gateway with type AND
52        // new Gateway(id, label, lane, pool, type)
53          temp = new Gateway(n.getID(), "", null,
54                             null, GatewayType.AND);
55          break;
56        case NT_AND_SPLIT:
57        // Splitting AND Gateway => create Gateway with type AND
58          temp = new Gateway(n.getID(), "", null,
59                             null,  GatewayType.AND);
60          break;
61        case NT_XOR_JOIN:
62        // Joining XOR Gateway => create Gateway with type XOR
63          temp = new Gateway(n.getID(), "", null,
64                             null,  GatewayType.XOR);
65          break;
66        case NT_XOR_SPLIT:
67        // Splitting XOR Gateway => create Gateway with type XOR
68          temp = new Gateway(n.getID(), "", null,
69                             null,  GatewayType.XOR);
70          break;
71        case NT_STARTLOOP:
72        // Loops are represented as XOR Gateways
73          temp = new Gateway(n.getID(), "", null,
74                             null,  GatewayType.XOR);
75          break;
76        case NT_ENDLOOP:
77          temp = new Gateway(n.getID(), "", null,
78                             null,  GatewayType.XOR);
79          break;
80        case NT_STARTFLOW:
```

```
81          // Start event => new Event of type START_EVENT
82          // new Event(id, name, lane, pool, type)
83            temp = new Event(n.getID(), n.getName(), null,
84                             null, EventType.START_EVENT);
85            break;
86          case NT_ENDFLOW:
87          // End Event
88            temp = new Event(n.getID(), n.getName(), null,
89                             null, EventType.END_EVENT);
90            break;
91        }
92        // Adds respective Lanes and Pools
93        // Searches in the List created for existing
94        // occurrences or creates new Lane/Pool
95        temp = setStaffProperties(temp);
96        // Out Element in HashMap for Arc creation later on
97        tasksNgates.put(temp.getId(), temp);
98        // Add Element to ProcessModel
99        p.addElem(temp);
100     }
101     // Arc creation: Iterate through all edges.
102     for(Iterator<StructuredEdge> i = edges.iterator();
103         i.hasNext();){
104       StructuredEdge edge = i.next();
105       // Create new Arc with source and destination Element.
106       // These elements are received from the previous
107       // created HashMap. As id, take numbers from
108       // Integer.MAX_Value downwards since Nodes are starting
109       // by zero.
110       // new Arc(id, label, source, target)
111       p.addArc(new Arc(id--, "",
```

```
112                 tasksNgates.get(edge.getSourceNodeID()),
113                 tasksNgates.get(edge.getDestinationNodeID())));
114     }
115     // Return the created ProcessModel
116     return p;
117 }
```

A.3. Source Code of NLTextAreaWidget

```
1  // Define the namespace
2  var NLTextAreaWidget = NLTextAreaWidget || {};
3
4  NLTextAreaWidget.NLTextAreaComponent = function (element) {
5   this.element = element;
6   this.element.innerHTML = "<div></div>";
7
8   var id = 0;
9   var isClickMode = true;
10
11  this.getId = function(){
12    return id;
13  };
14
15  this.setId = function(_id){
16    id = _id;
17  };
18
19  this.setClickMode = function(bool){
20          isClickMode = bool;
21  };
```

```
22  // Getter and setter for the value property
23  this.getText = function () {
24    return this.element.
25    getElementsByTagName("div")[0].innerHTML;
26  };
27
28  this.setText = function (value) {
29    this.element.getElementsByTagName("div")[0].innerHTML =
30    value;
31  };
32
33  this.setReadOnly = function(bool){
34    var root = this.element.getElementsByTagName("div")[0];
35    var component = this;
36    if(bool == true){
37      root.onclick="";
38      root.oncontextmenu="";
39      root.contentEditable = false;
40      component.updateText();
41    }
42    else{
43      if(isClickMode){
44            root.onclick = function () {
45              if(event.target.tagName =="P"
46                 ||event.target.tagName =="SPAN"){
47                    component.setId(event.target.id);
48                    if(event.ctrlKey)
49                      component.click("ctrl");
50                    else
51                      component.click("none");
52              }
```

```
53            };
54            root.oncontextmenu = function(){
55               if(event.target.tagName == "P"
56                  ||event.target.tagName =="SPAN"){
57                     component.setId(event.target.id);
58                     component.click("right");
59               }
60             };
61      }
62      else{
63        root.contentEditable = true;
64      }
65      component.updateText();
66    }
67  };
68  // Default implementation of the click handler
69  this.click = function (opt) {
70    alert("Error: Must implement click()");
71  };
72
73  this.updateText = function(){
74        alert("Error: Implement updateText()");
75  };
76
77  this.colorSelectedTasks = function(toColor){
78        var tasks = this.element.getElementsByTagName("P");
79        if(toColor == null){
80        for(var i = 0; i< tasks.length; i++){
81          tasks[i].style.background=null;
82        }
83    }
```

```
84      else{
85        for(var i = 0; i< tasks.length; i++){
86                  for(var k = 0; k < toColor.length; k++){
87                if(tasks[i].id == toColor[k]){
88                  tasks[i].style.background="#a9a9a9";
89                  break;
90                 }
91                else{
92                  tasks[i].style.background=null;
93                 }
94               }
95        }
96     }
97   };
98
99   this.colorSelectedDataElements = function(toColor){
100    var datas = this.element.getElementsByTagName("SPAN");
101    if(toColor == null){
102          for(var i = 0; i< datas.length; i++){
103         datas[i].style.background=null;
104       }
105    }
106    else{
107      for(var i = 0; i< datas.length; i++){
108            for(var k = 0; k < toColor.length; k++){
109              if(datas[i].id == toColor[k]){
110                if(datas[i].className == 'read'){
111                  datas[i].style.background="#00FF33";
112                 }
113                else{
114                  datas[i].style.background="#FF3300";
```

```
115                   }
116                   break;
117                 }
118                 else{
119                   datas[i].style.background=null;
120                 }
121               }
122       }
123     }
124   };
125 };
```

A.4. Source Codes of TextInvestigator Class Operations

```
1  // This method returns the resource of a setnence.
2  // The sentence must be tagged by the Stanford parser.
3  private String extractResourceFromTaggedSentence(
4          String sentence){
5    String resource = "";
6    //First,check if sentence is in passive voice
7    if(!this.isPassiveVoice(sentence)){
8      // If sentence is in active voice,
9      // the resource is behind the last verb.
10     String toCheck = sentence.substring(
11             sentence.lastIndexOf("/VB")+4);
12     String[] words = toCheck.split(" ");
13     // Check each word
14     for(String word: words){
15       // If the word is a proper noun, it is a resource
16       if(word.contains("/NNP")){
```

```
17          // Add new resource to the other parts,
18          // separate words with a blank
19          resource += word.replace("/NNP", " ");
20        }
21        // A word tagged as noun is also part of the resource
22            else if(word.contains("/NN")){
23              resource += word.replace("/NN", " ");
24            }
25          }
26    }
27    // Passive voice means, that the resource is in front of
28    // the first verb.
29    else{
30          // Get the part of the sentence in front of the
31          // first verb.
32          String toCheck = sentence.subSequence(
33             0, sentence.indexOf("/VB")).toString();
34          String[] words = toCheck.split(" ");
35          // Check like in other case.
36          for(String word: words){
37            if(word.contains("/NNP")){
38              resource += word.replace("/NNP", " ");
39            }
40            else if(word.contains("/NN")){
41              resource += word.replace("/NN", " ");
42            }
43          }
44    }
45    // Return the resource
46    return resource;
47  }
```

```
48 // This method returns the action of a sentence.
49 // The setnence must be tagged by the Stanford parser.
50 private String extractActionFromTaggedSentence(String sentence){
51   String action = "";
52   // If sentence is in passive voice, then the action is a
53   // past participle verb
54   if(this.isPassiveVoice(sentence)){
55     String[] words = sentence.split(" ");
56     // Check each word of sentence
57     for(String word: words){
58       // 'VBN' means the word is a verb, past participle
59       if(word.contains("/VBN"))
60         // Add the basic form of the verb to the action.
61         // But first get rid of the tag.
62         action += labelHelper.getInfinitiveOfAction(
63         word.replaceAll("/VBN", ""))+" ";
64       }
65     }
66     // If sentence is in active voice,
67     // then the action is a verb is 3rd. person present
68     // Execution is the same as before, but check for 'VBZ' tag.
69     else{
70       String[] words = sentence.split(" ");
71       for(String word: words){
72         if(word.contains("/VBZ"))
73           action += labelHelper.getInfinitiveOfAction(
74           word.replaceAll("/VBZ", ""))+" ";
75       }
76     }
77   // Return the action.
78   return action;
```

```
79 }
80 // This method check for passive voice in sentences.
81 // Sentence must be tagged first by Stanford parser.
82 private boolean isPassiveVoice(String taggedSentence){
83   String[] words = taggedSentence.split(" ");
84   for(int i = 0; i < words.length-1; i++){
85    /* If a word is a verb, 3rd person present,
86     * and the following word is a past participle verb
87     * and the current word is an inflection of 'be'
88     * this sentence must be passive voice.
89     */
90     if(words[i].contains("/VBZ") && words[i+1].contains("/VBN")
91        && (words[i].contains("is")|| words[i].contains("are"))){
92       return true;
93     }
94   }
95   return false;
96 }
```

List of Figures

List of Tables

Bibliography

[1] Stefan Büringer. Development of a Business Process Abstraction Component based on Process Views. 2012.

[2] John Sweller. How the human cognitive system deals with complexity. *Handling complexity in learning environments: Theory and research*, pages 13–25, 2006.

[3] Jens Kolb, Klaus Kammerer, and Manfred Reichert. Updatable Process Views for User-centered Adaption of Large Process Models. In *10th Int'l Conference on Service Oriented Computing (ICSOC'12)*, number 7636 in LNCS, pages 484–498. Springer, October 2012.

[4] Jens Kolb, Benjamin Rudner, and Manfred Reichert. Towards Gesture-based Process Modeling on Multi-Touch Devices. In *1st Int'l Workshop on Human-Centric Process-Aware Information Systems (HC-PAIS'12)*, number 112 in LNBIP, pages 280–293. Springer, June 2012.

[5] Jens Kolb and Manfred Reichert. Supporting Business and IT through Updatable Process Views: The proView Demonstrator. In *ICSOC'12, Demo Track of the 10th Int'l Conference on Service Oriented Computing*, number 7759 in LNCS, pages 460–464. Springer, March 2013.

[6] Leopold, H. (2013): Natural Language in Business Process Models, PhD Thesis, Humbolt University of Berlin.

[7] Henrik Leopold, Jan Mendling, and Artem Polyvyanyy. Generating natural language texts from business process models. In *Proceedings of the 24th international*

conference on Advanced Information Systems Engineering, CAiSE'12, pages 64–79, Berlin, Heidelberg, 2012. Springer-Verlag.

[8] Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. The refined process structure tree. In Marlon Dumas, Manfred Reichert, and Ming-Chien Shan, editors, *Business Process Management*, volume 5240 of *Lecture Notes in Computer Science*, pages 100–115. Springer Berlin Heidelberg, 2008.

[9] George A. Miller. Wordnet: a lexical database for english. *Commun. ACM*, 38(11):39–41, November 1995.

[10] Dan Klein and Chris Manning. Fast Exact Inference with a Factored Model for Natural Language Processing. In *Advances in Neural Information Processing Systems 15 (NIPS)*, 2002.

[11] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of English: the penn treebank. *Comput. Linguist.*, 19(2):313–330, June 1993.

[12] Igor A. Mel'čuk and Alain Polguère. A formal lexicon in the Meaning-Text Theory: (or how to do lexica with words). *Comput. Linguist.*, 13(3-4):261–275, July 1987.

[13] Benoit Lavoie and Owen Rambow. A fast and portable realizer for text generation systems. In *Proceedings of the fifth conference on Applied natural language processing*, ANLC '97, pages 265–268, Stroudsburg, PA, USA, 1997. Association for Computational Linguistics.

[14] HTML5 Introduction. http://www.w3schools.com/html/html5_intro.asp. -24.07.2013.

[15] JavaScript Tutorial. http://www.w3schools.com/js/default.asp. -24.07.2013.

[16] Vaadin API. https://vaadin.com/api/. -24.07.2013.

[17] Java SE7 API. http://docs.oracle.com/javase/7/docs/api/. -24.07.2013.

[18] Bruno Lowagie. *iText in Action: Creating and Manipulating PDF*. Manning Publications, 1 edition, December 2006.

[19] Fabian Friedrich, Jan Mendling, and Frank Puhlmann. Process Model Generation from Natural Language Text. In Haralambos Mouratidis and Colette Rolland, editors,

Advanced Information Systems Engineering, volume 6741 of *Lecture Notes in Computer Science*, pages 482–496. Springer Berlin Heidelberg, 2011.

[20] Camille Ben Achour. Guiding scenario authoring. In 8th European-Japanese Conference on Information Modelling and Knowledge Bases, pages 152-171. IOS Press, 1998.

[21] Vaadin 7 Loves JavaScript Components. https://vaadin.com/blog/-/blogs/vaadin-7-loves-javascript-components. -24.07.2013.

[22] Vaadin Context Menu Addon. https://vaadin.com/directory#addon/contextmenu. -24.07.2013.

[23] JTidy. http://jtidy.sourceforge.net/. -24.07.2013.

Name: Wolfgang Wipp Matrikelnummer: 698982

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Wolfgang Wipp