e-Informatica Software Engineering Journal, Volume 3, Issue 1, 2009

Automated Code Generation from System Requirements in Natural Language

Jan Franců*, Petr Hnětynka*
*Faculty of Mathematics and Physics, Department of Software Engineering, Charles University in Prague
jfrancu@gmail.com, hnetynka@dsrg.mff.cuni.cz

Abstract

An initial stage of a software development is specification of the system requirements. Typically, these requirements are expressed in UML and consist of use cases and domain model. A use case is a sequence of tasks, which have to be performed to achieve a specific goal. The tasks of the use case are written in a natural language. The domain model describes objects used in the use cases. In this paper, we present an approach that allows automated generation of executable code directly from the use cases written in a natural language. Usage of the generation significantly accelerates the system development, e.g. it makes immediate verification of requirements completeness possible and the generated code can be used as a starting point for the final implementation. A prototype implementation of the approach is also described in the paper.

1. Introduction

Development of software is covered by several stages from which one of the most important is the initial stage – collecting system requirements. These requirements can be captured in many forms, however, use of the Unified Modeling Language (UML) has become an industry standard at least for large and medium-size enterprise applications. Development with UML [8] is based on modeling the developed system at multiple levels of abstraction. Such a separation helps developers to reflect specific aspects of the designed system on different levels and therefore to get a "whole picture" of the system.

Development with the UML starts with definition of goals of the system. Then, main characteristics of the system requirements are identified and described. A behaviour of the developed system is specified as a set of *use cases*. A *use case* is a description of a single task performed in the designed system [3]. The task itself is further divided into a sequence of steps that are performed by communicating entities. These entities are either parts of the system or users of the system. A step of a use case is specified by natural language sentences. The use cases of the system are completed by a *domain model* that describes entities, which together form the designed system and which are referred to in the use cases.

Bringing a system from the design stage to the market is a very time-consuming and also money-consuming task. A possibility to generate an implementation draft directly from the system requirements would be very helpful for both requirement engineers and developers and it would significantly speed up development of the system and decrease time required to deliver the system to the market and also decrease amount of money spent. The system use cases contain work-flow information and together with the domain model capture all important information and therefore seem to be sufficient for such a generation. But the problem is that the use cases are written in a natural language and there is a gap to overcome to generate the system code.

1.1. Goals of the paper

In this paper, we describe an approach, which allows to generate an implementation of a system from the use cases written in a natural language. The process proposed in the paper enables software developers to take an advantage of the carefully written system requirements in order to accelerate the development and to provide immediate feedback for the project's requirement engineers by highlighting missing parts of the system requirements. The process fits in the incremental development process where in each iteration developers can eliminate shortcomings in design. In addition, the process can be customized to fit in any enterprise application project.

Described approach is implemented in a proof-of-the-concept tool and tested on a case study.

To achieve the goals, the paper is structured as follows. Section 2 provides an overview of the UML models and technologies required for use case analysis. Section 3 shows how our generation tool is employed in the application development process. Section 4 describes the tool and all generation steps in detail while Section 5 presents particular examples of the generated code. Section 6 evaluates our approach and the paper is concluded in Section 7, where future plans are also shown.

2. Specification of Requirements

The *Unified Modeling Language* (UML) is a standardized specification language for the software development. Development with UML is based on modeling a system at multiple levels of abstraction in separated models. Each model represented as a set of documents clarifies the abstraction on a particular level and captures different aspects of the modeled system. The UML-based methodologies standardize whole development process and ensure that the designed system will meet all the requirements. UML also increases possibilities to reuse existing models and simplifies reuse of code. The developers can use several existing modeling tools/frameworks (e.g. [10]) to support this process.

In this paper, we work with the UML documents created during the initial stage of the system development, i.e. with the requirement specification. Results of the stage are captured in *use cases* and *domain model*.

2.1. Use Cases

A *use case* in the context of UML is a description of a process where a set of entities cooperates together to achieve a goal of the use case. The entities in the use case can refer to the whole system, parts of the system, or users. Each use case has a single entity called *system under discussion* (*SuD*); from the perspective of this entity, the whole use case is written. An entity primarily communicating with *SuD* is called a *primary actor* (*PA*). Other entities involved in the use case are called *supporting actors* (*SA*).

Each use case is a textual document written in a natural language. The book [3] recommends the following structure of the use case: (1) header, (2) main scenario, (3) extensions, and (4) sub-variations.

The header contains the name of the use case, *SuD* entity, primary actor and supporting actors. The main scenario (also called the *success scenario*) defines a list of steps (also called *actions*) written as sentences in a natural language that are performed to achieve the goal of the use case. An action can be extended with a *branch action*, which reflects possible diversions from the main scenario. There are two types of the branch actions: *extensions* and *sub-variations*. In an *extension*, actions are performed in addition to the extended action, while in a *sub-variation*, actions are performed instead of the extended action. The first sub-action of a branch action is called a *conditional label* and describes necessary condition under which the branch action is performed.

The above described structure is not the only possible one – designers can use any structure they like. In our approach, we assume the use cases satisfy these recommendation as it allows

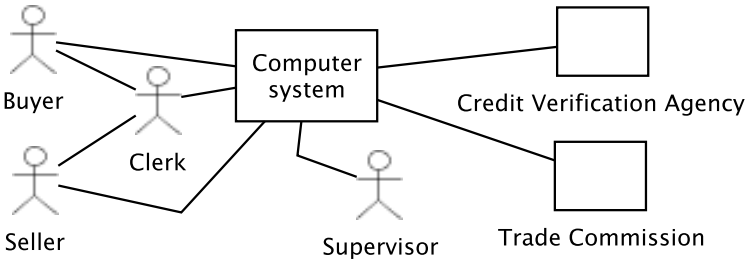

Figure 1. The Marketplace project entities

UseCase: Buyer buys a selected item
SuD: Clerk
PA: Buyer
Supporting actor: Computer System

Main success scenario specification:
1 Buyer submits to the clerk a reference to a selected offer.
2 Clerk submits the reference to the system.
3 Clerk reports the system response to the seller and requests billing and shipping information, payment method and payment details.
4 Buyer submits to the clerk the requested billing and shipping information, payment method and payment details.
5 Clerk enters the billing and shipping information, payment method and payment details.
6 Clerk reports the system response (with the unique acknowledgment) to the buyer.

Extensions:
3a System failed to validate the offer.
3a1 Use case abort.

Figure 2. Use case example

us to process the use case automatically and generate the system implementation. Such an assumption does not limit the whole approach in a significant way, hence the book [3] is widely considered as a "bible" for writing the use cases (in addition, we already have an approach for using use cases in fact with any structure – see Sect. 7).

In the rest of the paper we use as an example a *Marketplace* project for on-line selling and buying. A global view of the application entities is depicted on Figure 1. There are several actors, which communicate with the system. *Seller*s enter offers to the system and *Buyer*s search for interesting offers. Both of them mainly communicate directly with the *Computer system* – in few cases, they have to communicate through a *Clerk* who passes information to the Computer system. There is also a *Supervisor* which maintains the Computer system. A *Credit verification agency* verifies Seller's and Buyer's operations and finally a *Trade commission* confirms the offers.

The use case on Figure 2 is a part of the Marketplace specification (the whole specification has 19 use cases) and it describes communication between the Buyer (as PA), Clerk (as SuD), and Computer system. It is prepared according to the recommendations.

2.2. Domain Model

A *domain model* describes entities appearing in the designed system. Typically, the domain model is captured as a UML class diagram and consists of three types of elements: (1) *conceptual classes*, (2) *attributes* of conceptual classes, and (3) *associations* among conceptual classes.

Conceptual classes represent objects used in the system use cases. The attributes are features of the represented objects and associations describe relations among the classes. Figure 3 shows the Marketplace domain model.

As described in [8], noun phrases appearing in the use cases can be used for determining class names during creation of the domain model (in further detail, such a relation between the class diagrams and use cases is investigated in [1]).

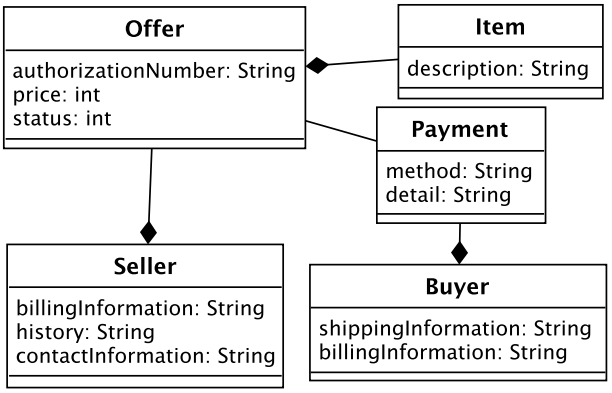

Figure 3. Marketplace domain model

2.3. Procasor Tool and Procases

The *Procasor* [6] is a tool for automated transformation of natural language (English) use cases into a formal behaviour specification. The transformations are described in [9] and further extended in [4] where almost all restrictions of a use case step syntax were removed.

As a formalism into which the natural language use cases are transformed the Procasor uses *procases* [9] that are a special form of *behaviour protocols* [14]. In addition to procases, a UML state machine diagram is also generated.

A procase is a regular expression-like specification, which can describe behaviour of a single entity as well as of the whole system [13]. The procases generate so called *traces* that represent all possible valid sequences of actions described by the use cases. Figure 7 shows a procase derived from the use case shown in Figure 2.

A procase is composed of operators (i.e. *+*, *;*), procedure calls (*{*,*}*), action tokens, and supporting symbols (i.e. round parenthesis for specifying operators' precedence). Each action token represents a single action that has to be performed and its notation is composed of several parts. First, there is a single character representing a type of the action. The possible types are *?* resp. *!* for request receiving resp. sending action, *#* for internal actions (unobservable by others than *SuD*) and *%* for special actions. The action type is followed by the entity name on which the action is performed. Finally, the name of the action itself is the last part (separated by a dot). In a case, there is no entity name, the action is internal. For example, *?B.submitSelectOffer* is the *submitSelectOffer* action where SuD waits for a request from the *B* (Buyer) entity.

The procases use the same set of operation as regular expressions. These are: *** for iteration, *;* for sequencing, and *+* for alternatives. In this paper, we call the *alternative* operator as a *branch action*, its operands (actions) as *branches*, and the *iteration* operator with its operand as a *loop action*.

A special action is *NULL* which means no activity and is used in places with no activity but the procase syntax requires an action specified there (e.g. with the alternative operator). Another special action is the first action inside a non-main scenario branch, which is alled *condition branch label* and expresses the condition under which the branch is triggered. Finally, the *%ABORT* special action represents a failure ending of the procase.

Procedure calls (written as a sequence of actions in curly brackets) represent a behaviour (mostly composed of inner actions) of the request receive action after which they are placed (the action is called *trigger action*).

2.4. Goals Revisited

As described in the sections above, the Procasor tool parses the use cases written in a natural language and generates a formal specification of behaviour of the designed system. A straightforward idea is then why to stop just with the generated behaviour description and not to generate also an implementation of the system which implements the work-flow captured in the use cases.

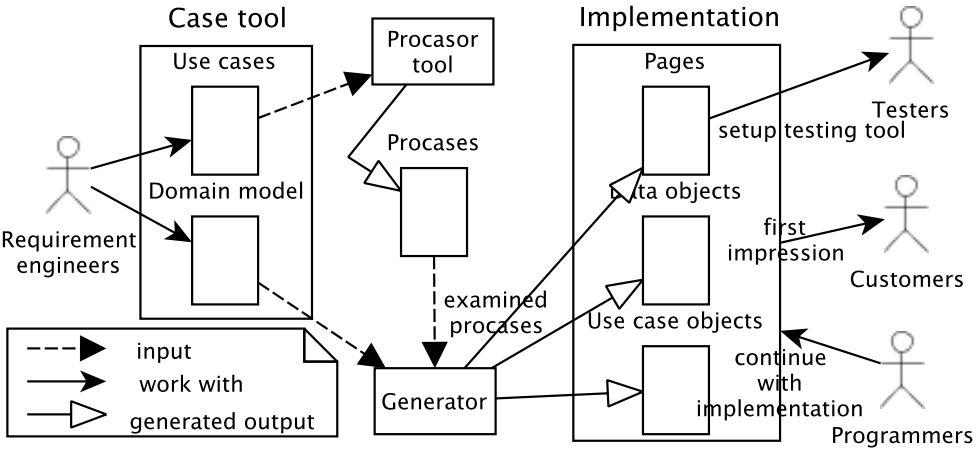

Figure 4. Development process overview

The goal of this paper is to present an extension of the Procasor tool that based on the use cases generates executable implementation of the designed system.

3. Generating Process

The development process with our generating tool is as follows. First, requirement engineers collect all requirements and describe them in the form of use cases. Then the Procasor tool automatically generates procases. In parallel, the requirement engineers create a project domain model. As a next step of validating the use cases, the generated procases can be reviewed. Then, our generator is employed and produces an implementation of the developed system. The generated implementation consists of three main parts: (i) use case objects where work-flow captured in a use case is generated, (ii) pages which are used to communicate with users of the system, and (iii) entity objects where the business logic is kept.

The generated implementation is only an initial draft and serves primarily for testing the use cases and domain model. But it can be also used as a skeleton for the actual implementation and/or to allow customers to gain first impressions of the application. The whole development process is illustrated in Figure 4.

At this point a common mistake has to be emphasized (which is also emphasized in [8]). The system requirements cannot be understood as final and unchangeable. Especially in incremental development, the requirements are created in several iterations and obviously the first versions are incomplete. Therefore if the generator is used on such input, it can generate a completely wrong implementation. But this implementation can be used to validate the use cases, repair them and regenerate the implementation.

4. Generating Tool in Detail

The generator of the implementation takes as an input the procases generated by the Procasor and the created domain model of the designed system. From these inputs, it generates the executable implementation.

The generation is automated and it consists of three steps:

1. First, procases generated from the Procasor are rearranged into a form, in which they still follow the procases syntax but are more suitable for generating the implementation (Sect. 4.1).
2. Then, a relation between words used in the use cases and elements in the domain model is obtained and parameters (i.e. their numbers and types) of the methods are identified (Sect. 4.2).
3. Finally, the implementation of the designed system is generated (Sect. 4.3).

4.1. Procase Preprocessing

The procases produced by the Procasor do not contain procedure calls brackets (see 2.3), which are crucial for successful transformation of the procases into the code. Except several marginal cases, each use case is a request-response sequence between SuD and PA (for enterprise applications). In the procase, a single request-response element is represented as a sequence of actions from which the first one is the request receive action (i.e. starts with ?) and then followed by zero or more other actions (i.e. sending request action, internal actions, etc.). In other words, SuD receives the request action, then performs a list of other actions, and finally returns the result (i.e. end of the initial request receive action). Hence, the sequence of actions after the request receive action can be modeled as a procedure content and enclosed in the procedure call brackets.

The following example is a simple procase in a form produced by the Procasor:

$$?PA.a; \#b; !SA.c; ?PA.d; \#e; \#f$$

After identifying the procedure calls, the procase is modified into the following form:

$$?PA.a\{\#b; !SA.c\}; ?PA.d\{\#e; \#f\}$$

At the end, the code generated from this procase consists of two procedures – first one generated from the *?PA.a* action and internally calling the procedures resulted from *#b* and *!SA.c*, and the second one generated from *?PA.d* and calling *#e* and *#f*.

The approach described in the paragraph above works fine except for several cases. In particular, these are: (1) first action of the use case is not a request receive action, (2) a request receive action is in a branch(es), and (3) a request receive action is anywhere inside a loop.

In the case when the first action of the use case is not a request receive action, a special action *INIT* is prepended to the use case and the actions till the first request receive action are enclosed in the procedure call brackets. In the generated code, a procedure generated from the *INIT* action is called automatically before the other actions.

Two other cases cannot be solved directly and require more complex preprocessing. To solve these cases, we have enhanced procases with so called *conditional events*, which allow "cutting" branches of the procase and arrange them in a sequence, but which do not modify the procase syntax.

The conditional events allow to mark branches of the alternatives by a boolean variable (written in the procase just as a name without any prefix symbol followed by a colon, e.g. D:). The variables can be set to *true* via the action written as the variable name prefixed with the $ symbol (e.g. $D) or to false by its name with the $ and ∼ symbols (e.g. ∼$D). At the beginning of each procase, all variables are undeclared.

These events modify the behaviour of the procase in a way that only traces containing the action, which sets the variable to *true*, continue with the branches marked by this variable. When the value of the variable is *false*, the traces continue with the unmarked branches as in unchanged procase.

4.1.1. Branch transformation

First, we show how to rearrange a procase with the request receive action placed in a branch. The general approach of identifying procedures as described above does not work as it would result in nested procedures. To avoid them, it is necessary to rearrange the procase in order to place affected branches sequentially.

We illustrate the branch transformation on the following example:

$$?a; \#b; (\#c; \mathbf{?d}; \#e + \#f; (\#g; \#h + \#i)); \#j$$

The problematic action is *?d* placed in a branch and the whole example is visualized in Fig. 5(a).

The approach of rearranging branches is as follows. Instead of the affected request receive action, the declaration of a conditional event variable is placed. The original action with all subsequent actions till the end of the branch are moved outside the alternative and marked with the chosen variable – depicted in Fig. 5(a ⇒ b).

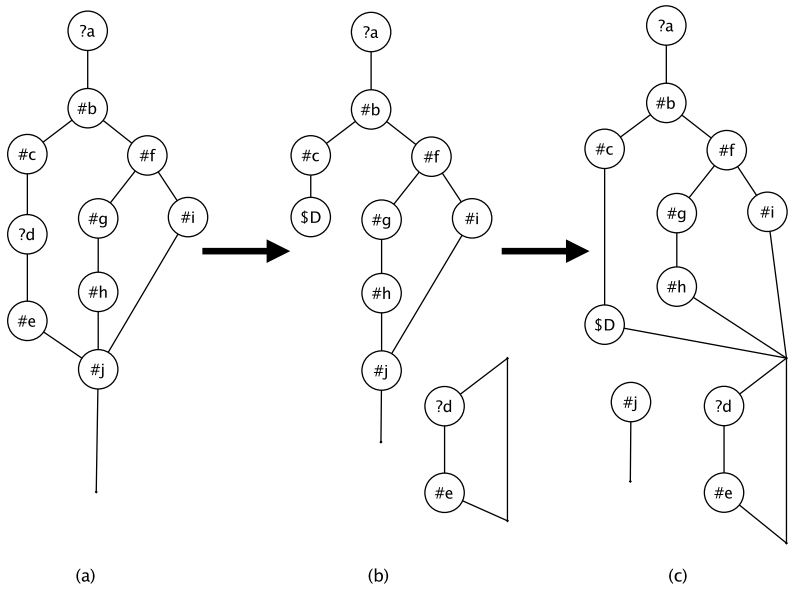

Figure 5. First part of the Branch transformation

The *NULL* action is added as a second branch of the newly created marked branch.

Now, the actions that followed after the original branch action (till the first request receive action) have to be appended to all other branches of this branch action except the branch with the variable declaration – depicted in Fig. 5(b ⇒ c). In addition, these actions are placed at the end of the newly created branch. In a case the variable declaration is placed in more than one branch (i.e. the request receive actions were in more branches), appending of subsequent actions (till the first request receive action) has to be done for all these branches and variables – depicted in Fig. 6(d ⇒ e). This appending guarantees that the resulting procase in Fig. 6(f) generates the same traces as the original one. Now, the general approach of identifying procedures can be applied and yields the following procase:

$$?a\ \{\#b; (\#c; \$D + \#f; (\#g; \#h; \#j + \#i; \#j))\};$$
$$(D : ?d\ \{\#e; \#j\} + NULL)$$

Another example is in Figure 7, which shows the procase of the use case in Fig. 2 that also contains problematic request receive action. Figure 8 depicts the procase after the branch transformation, i.e. it is completely equivalent to the former one and does not contain the problematic branch.

4.1.2. Loop transformation

The transformation of the procases with the request receive action located in a loop action is quite similar to the previous case. Again, the transformation guarantees that the resulting procase generates the same traces as the original one.

The following procase is an example with the request receive action inside the loop:

$$?PA.a; \#b; (\#c; \mathbf{?PA.d}; \#e) * \#f$$

And the resulting transformed procase:

$$?PA.a\ \{\#b; (\#c; \$D + \#f)\};$$
$$(D : ?PA.d\ \{\#e; (\#c + \#f; \sim \$D)\} + NULL)*$$

4.1.3. Unresolved cases

In a case the request receive action is located in two or more nested loops or in a loop nested in branches, the previous two transformations do not work. The procase is then marked as unresolved, excluded from the further processing and has to be managed manually. On the other hand, such use cases are very unreadable (see [8] for suggestions about avoiding extensions of extensions or complex nested loops which results into this problematic procases) and therefore the skipped use cases are candidates for rewriting in a more simple and readable way.

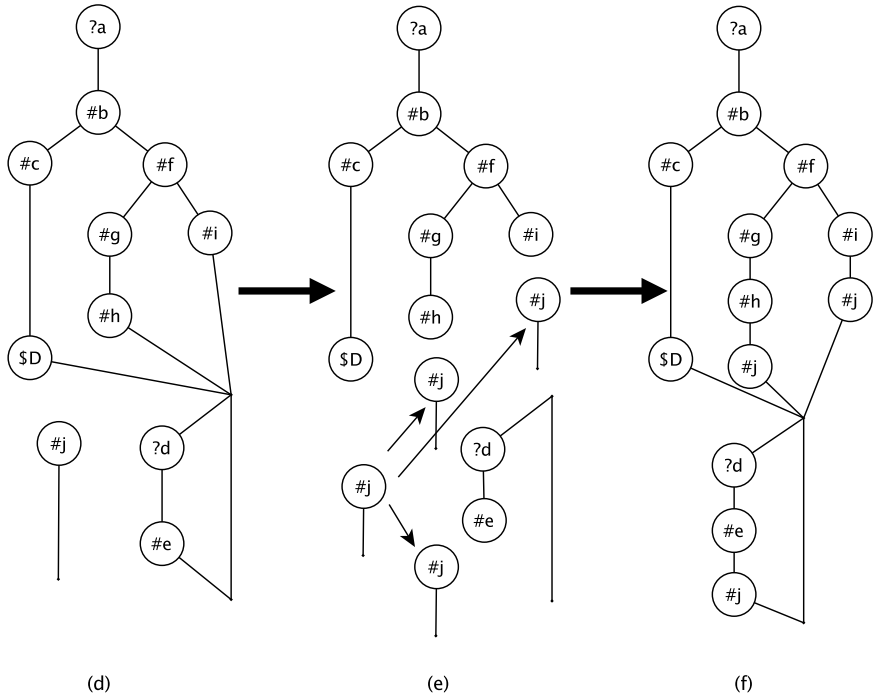

Figure 6. Second part of the Branch transformation

```
?B.submitSelectOffer;
!CS.submitSelectOffer;
!B.reportSystemResponse;
(
  ?B.submitBillingShippingInformationPaymentMethodPaymentDetail;
  !CS.enterBillingShippingInformationPaymentMethodPaymentDetail;
  !B.reportSystemResponse
+
  #validateSystemFail;
  %ABORT
)
```

Figure 7. Procase example before the branch transformation

4.2. Determining Arguments

Once the procases have been preprocessed into sequences of actions grouped as procedure calls, the next step is to determine arguments of the identified procedures, types of these arguments, and how their values are assigned. The arguments are subsequently used as arguments for methods in the final generated code.

In our approach, we are using a fact mentioned in [8] that noun phrases appearing in the use cases are directly related with the domain model elements names. The process of determining arguments is as follows.

First, all noun phrases (which may refer to the data manipulated in the use case step) are extracted by the Procasor from the use case step sentence. In addition, we also take into account *verbs* from the sentence as they can refer to the relations between the conceptual classes in the domain model.

The list of extracted words is then matched against keywords of the domain model (by the keywords we mean names of the classes, attributes, and associations) in order to discover which words are actual attributes and to obtain their types. There are many options to match the keywords – currently in our implementa-

```
?B.submitSelectOffer {
  !CS.submitSelectOffer;
  !B.reportSystemResponse;
  (
    #validateSystemFail;
    %ABORT
  +
    $SBSIPMPD
  )
};
(SBSIPMPD:
  ?B.submitBillingShippingInformationPaymentMethodPaymentDetail {
    !CS.enterBillingShippingInformationPaymentMethodPaymentDetail;
    !B.reportSystemResponse1
  }
+
  NULL
)
```

Figure 8. Procase example from Fig. 7 after the branch transformation

tion we use a simple case-insensitive equality of strings. Such a matching approach can be seen as insufficient but on the other hand, projects commonly follow a chosen terminology (many times explicitly captured in the requirement documents) and therefore our approach is satisfactory in most of the cases.

The determined arguments are compared (by the name and type) with arguments of previous procedures (if they exist) and the already used arguments are copied (their values). If the previous procedures are located in a branch parallel with the *NULL* action they are excluded from processing as they may not be called before the processed one.

Now, the process behaves differently based on a type of the entity, on which the action is called. The types are (i) human user entities (*UE*) such as buyer, seller, etc. and (ii) parts of the system or other computer systems (i.e. system entity – *SE*).

For the trigger action and actions with UE SuD, the unmatched determined types are used as arguments (i.e. parameters which have to be inputted by users). For actions with SE SuD the unmatched determined types are also added as arguments but with default values (during the development of the final application, developers have to provide correct values for them).

4.3. Generating Application

Structure of the generated application employs multiple commonly used design patterns for enterprise applications. Based on these patterns, the generated code is structured into three layers – presentation layer, middle (business) layer, and data layer. In the following text, we refer to objects of the presentation layer as *pages* because the most commonly used presentation layer in contemporary large applications employs web pages, but any type of the user interface can be generated in a similar way.

The middle layer consists of so called *use case objects* which contain the business logic of the application. Also, the middle layer contains *entity objects* where the internal logic (implementation of the basic actions) is generated. The use case objects implement the ordering of the actions and call the entity objects.

We do not describe generation of the data layer, as it is well captured in common UML tools and frameworks (generation of classes from class diagrams etc. – see Sect. 6).

The generation depends on the type of entity – pages are generated for UE while for SE a non-interactive code only. Thus, a page is generated for every action performed by UE (procedure call triggering actions and procedure call internal actions).

If the use case has SuD as UE then elements generated from the actions located inside a procedure call are named with the suffix "X" to allow their easier identification during future development, as in most cases they have to be modified by developers.

Based on combination of the communicating actors (UE vs. SE), the generation distinguishes four cases how the code is generated from a procedure call:

1. If PA and/or SA is UE, then a page is generated for every procedure call triggering action, which is triggered by this UE.
2. If PA and/or SA is SE, an action implementation method is generated in the actor entity object and the action method body contains a call to the corresponding use case object.
3. If SuD is UE, then a method in the corresponding use case object is generated for each procedure call of SuD. The method body calls the actor entity object and redirects to "X" pages, which manage the internal procedure call actions. Internal procedure call actions are generated in a similar way to the request receive action with UE PA – the "X" page and a method inside the "X" use case object are generated. The method inside the "X" use case object is generated as a simple delegation method to the corresponding entity object and redirection to the particular page.
4. And finally if SuD is SE, then inside the corresponding use case object, a method with the body containing the internal procedure call actions is generated.

Figure 9 shows the procase of the Clerk-buys-selected-Offer-on-behalf-of-Buyer use case and Figure 10 overviews all generated elements from the use case.

The following sections describe each type of the generated objects in more details.

4.3.1. Pages

As generated, pages are intended for testing the use cases and are expected to be reimplemented during the further development. A single page is generated for each action interacting with UE. In a case of UE PA, there is a page for every triggering action and, in addition for UE SuD, there is also a page for every procedure call internal action. If the action has arguments which can be inputted then for each of them an input field is generated. Values are assigned by humans during testing of the generated system.

For the UE PA actions, the corresponding pages have a button (an input control element) that allows to continue to the next page, i.e. to continue in the use case (there is only a single button as there is no other choice to continue). On the pages belonging to the UE SuD actions, there are several buttons, which reflect the possibilities of continuation in the original use case. For a sequence of the actions, the page contains the "continue" button; if the next action is a branch action then the page contains a button for each branch (the default button is for the main scenario branch – the buttons for the rest of the branches are labeled by the branch condition label; if the next action is a loop action then the page has a button to enter the loop and another button to skip the loop (following the definition of loop operation).

4.3.2. Use Case Objects

The use case objects contain the business logic (work-flow) of the use case, i.e an order of actions in the main scenario and all possible branches. Bodies of the generated methods differ according to SuD.

UE SuD: As described above, a method in the corresponding use case object is generated for each procedure call of UE SuD. For each trigger action, the method body contains a call to the particular entity object and redirection to a page of the subsequent action. For internal procedure call actions, a similar method body is created in "X" use case object.

SE SuD: A body of the method generated for the procedure call trigger action contains the internal procedure calls. For the SuD internal actions, methods are called on the use case SuD entity object and for request send actions, meth-

```
?CL.submitItemDescription {
    (
        #priceAssessmentAvailable;
        !Sl.providePriceAssessment
    +
        #validateDescription;
        (
            #validationPerformedSystemFails;
            %ABORT
        +
            NULL
        )
    )
};
?CL.enterPriceContactBillingInformation {
    #validateContactInformation;
    !SU.validateSeller
};
?SU.permitSeller {
    !TC.validateOffer;
    (
        #listOffer;
        !Sl.respondUniquelyIdentifiedAuthorizationNumber
    +
        #tradeCommissionRejectsOffer;
        %ABORT
    )
}
```

Figure 9. Clerk-buys-selected-Offer-on-behalf-of-Buyer use case

ods are called on the action triggered entity objects.

The branch actions are generated as a sequence of the condition statements (i.e. *if () ... else if () ...*) with as many elements as branches in the branch action. In each *if* statement, particular actions are generated, while the last *else* statement contains the main scenario actions. A similar construction but with a loop statement (*while*) is created for the loop action.

The number of iteration in the loop statement and choice of the particular branch in the condition statements cannot be determined from the use case. Therefore, the statements are generated with predefined but configurable constants inside the use case object.

4.3.3. Entity Objects

The internal logic of actions is not captured by the use cases neither by the domain model. Therefore, the entity objects are generated with almost empty methods containing only calls to a logger and they have to be finished by developers. For testing purposes, the logging methods seem to be the most suitable ones as designers can immediately check the traces of the generated system.

4.4. Navigation

Navigation (transitions) between the pages is an important part of the application internal logic as it determines the part of the system work flow (the order of procedure calls and sequence of actions). The navigation is derived from the procases as a set of navigation rules. The pages/objects have associated these rules that contain under which circumstances a transition has to be chosen.

In general, the navigation rules are created from the branch actions, loops, and special ac-

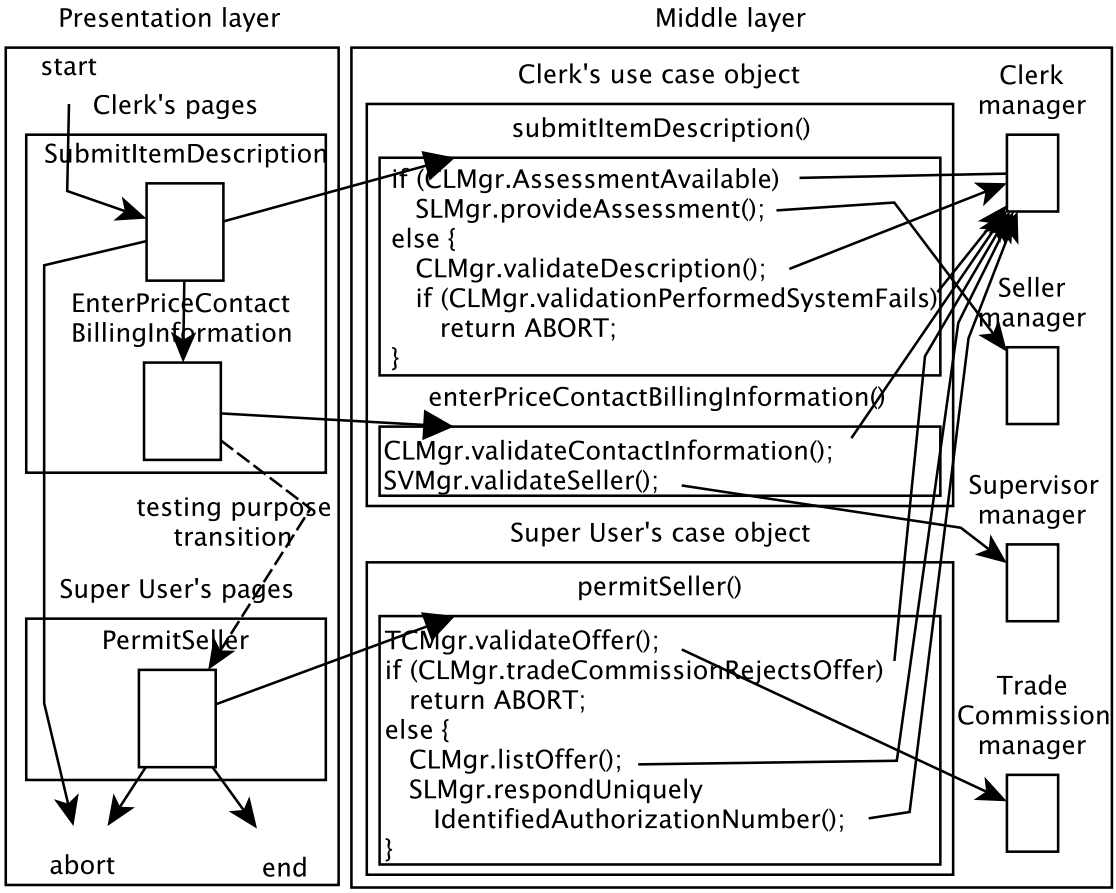

Figure 10. Overview of elements generated from the procase in Fig. 9

tions that can change transitions (aborts, etc.). In a case of the use case with SE SuD, the rules are applied to determine transitions between calls of the actions. In a case of UE SuD, the rules determine how the pages are generated, i.e. which buttons are placed on them.

5. Generated Application Example

To prove that our approach is feasible, we have implemented the proposed generator. As a particular technology for the generated applications, we have chosen the Java EE platform with Enterprise Java Beans (EJB) as the business layer and Java Server Faces (JSF) as the presentation layer. These technologies have been chosen as they are commonly used for large enterprise applications today and thus they can be used as starting point for continuing the application implementation. The generator itself has been written in plain Java.

The generator produces code together with an Ant build file, which can immediately compile and deploy the application to the JBoss application server [7], which allows users to inspect and modify code and iteratively test the application.

Based on the chosen technologies and used design patterns the first two application layers are mapped into the following five tiers. The pages results in two tiers: (i) JSF pages and (ii) backing beans (BB). The middle layer then results into three tiers: (iii) business delegator tier (BD), (iv) Enterprise Java Bean tier (EJB), and finally (v) manager tier (MGR). Generation of the data persistence layer is not currently supported but it is a simple task, which we are plan to add soon (see Sect. 7).

In the rest of this section, we describe in more detail all the generated elements produced from a single use case. As a particular example, the *Clerk submits an offer on behalf of a Seller* use case from the Marketplace example is used. It has the following content.

UseCase: Clerk submits an offer on behalf of a Seller (part)
SuD: Computer System
PA: Clerk

Main success scenario:
1 Clerk submits information describing an item.
2 System validates the description.

Extensions:
2a Validation performed by the system fails.
2a1 Use case aborted.

Sub-variations:
2b Price assessment available.
2b1 System provides the seller with a price assessment.

The Procasor and the preprocessing step of our generator produce the following procase:

```
?CL.submitItemDescription {
    (
        #priceAssessmentAvailable;
        !Sl.providePriceAssessment
    +
        #validateDescription;
        (
            #validationPerformedSystemFails;
            %ABORT
        +
            NULL
        )
    )
}
```

5.1. JSF Pages

All action pages are generated as described in 4.3.1. To allow easy testing, each page displays the use case together with the corresponding procase – both with the highlighted currently processed action and acting entity. The following listing shows a core of the generated JSF page for the use case above.

```
<h:form>
  <h:outputText value="itemDescription : " />
  <h:inputText id="itemDescription"
    value="#{ComputerSystem_
          ClerkSubmitsAnOfferOnBehalfOfASeller_
          ClerkBBean.itemDescription}" />
  </br>
  <h:outputText value="sellerBillingInformation : "  />
  <h:inputText id="sellerBillingInformation"
    value="#{ComputerSystem_
          ClerkSubmitsAnOfferOnBehalfOfASeller_
          ClerkBBean.sellerBillingInformation}" />
  </br>
  <h:commandButton value="submitForm"
    action="#{ComputerSystem_
          ClerkSubmitsAnOfferOnBehalfOfASeller_
          ClerkBBean.submitItemDescription}" />
</h:form>
```

The JSF page has a simple form with an input field for the action argument and a submit button. The triggering action **submitItemDescription** has an argument **itemDescription** which is bound to the use case backing bean variable.

5.2. Backing Beans

According to the JSF framework, the pages are supported by backing beans, which are Java classes containing all variables that can be set by the pages and handling all actions possibly activated by the pages. For the variables, BBs provide setter/getter methods. Also, BBs provide methods for calls to the next tier – business delegators.

The following listing shows the BB's method, which is called when a user submits the form on the page.

```
public String submitItemDescription() {
  try {
    return computerSystem_
      ClerkSubmitsAnOfferOnBehalfOfASellerBD
      .submitItemDescription(getSessionObject()
      . getSellerBillingInformation (), itemDescription,
      getSessionObject());
  } catch (DelegateException e) {
    e.printStackTrace();
  }
  return "abort";
}
```

Before the method call starts, the value of the form's input field is automatically set via the BB's setter method. The value is then used in the method as a parameter of the call to the BD tier.

5.3. Business Delegator

Business delegators are Java classes created on the basis of the Business Delegate pattern [17]. In the generated application, each use case has its own BD, which provides calls to the use case EJBs. Internally, methods of BDs use the Service Locator pattern [17] to locate EJBs.

The method shown in the listing only delegates calls to the use case EJB. The **getBean** method contains code for obtaining a use case bean **LocalHome** interface, creating the bean, and returning the use case bean stub. The stub is then used for the actual call.

```
public String submitItemDescription(String
  sellerBillingInformation,
  String itemDescription, ComputerSystem_
  ClerkSubmitsAnOfferOnBehalfOfASellerSO
  sessionObject) {
  try {
    return getBean().submitItemDescription(
      sellerBillingInformation,
      itemDescription, sessionObject);
  } catch (BeanException e) {
    throw new DelegateException(
      this.getClass().getName() +
      ".submitItemDescription()", e);
  }
}
```

5.4. EJB

The use case objects are generated as stateless session beans. There is no change against the general process described in 4.3.2.

The shown EJB method contains internal logic, i.e. actions located inside the triggered procedure call are executed in this method. The method body structure corresponds to the procase procedure call. Each action is generated as method delegating call to the particular MGR.

```
public String submitItemDescription(String
  sellerBillingInformation,
  String itemDescription, ComputerSystem_
  ClerkSubmitsAnOfferOnBehalfOfASellerSO
  sessionObject) {
  if (Constants.computerSystem_
    ClerkSubmitsAnOfferOnBehalfOfASeller_
      priceAssessmentAvailable) {
    getSellerManager().providePriceAssessment();
  }
  else {
    getComputerSystemManager()
      .validateDescription(itemDescription);
    if (Constants.computerSystem_
      ClerkSubmitsAnOfferOnBehalfOfASeller_
        validationPerformedSystemFails) {
      return NavigationConstants.ABORT;
    }
  }
  return NavigationConstants.CONTINUE;
}
```

5.5. Entity Managers

The entity objects are generated as entity manager classes and they are accessed from the beans, again using the Service locator pattern [17]. The methods of the managers print logs to a console (as explained in Section 4.3.3). We do not show here the generated MGR as its methods contain only these logger calls.

5.6. Additional Elements

In addition to the described elements, there are several additional generated objects that are used across the tiers. Namely, they are *Value objects* and *Session objects*. The former ones are generated and used for each type of arguments of the use case actions, while the later ones hold the value objects among different calls in a use case.

The *INIT* procedure call is generated as the **init** method of the corresponding use case EJB. The special action *%ABORT* is not modeled as an exception but rather as a predefined constant returned from the particular methods.

6. Evaluation and Related Work

To verify our generator, we used it on the Marketplace application described in Sect. 2.1. The generated implementation was compiled and directly deployed to an application server. The implementation consists of approx. 70 classes with 13 EJBs. The complete application has 92 action, from which only 16 actions were generated with wrong arguments and had to be repaired manually (we are working on an en-

hanced method of argument detection – see Sect. 7).

Testing of the generated application discovered a necessity to add one use case, two missing extensions in another use case, and also suggested restructuring other two use cases. All these defects could be detected directly from the use cases but with generated application, they became evident immediately.

Our tool can be also viewed as an ideal application of the Model-driven Architectures (MDA) [11] approach. In this view, the uses cases and domain model serve as a platform independent model, which via several transformations are transformed directly into an executable code, i.e. platform specific model.

Currently, the existing tools usually generate just data structures (source code files, database tables, or XML descriptors) from the UML class diagrams but no interaction between entities (i.e. they handle just the class diagrams) and as far as we know, there is no tool/project that generates the implementation from the description in a natural language. Below, there are several projects or tools that take as an input not only class diagrams but still they work with diagrams and not with a natural language.

The AndroMDA [2] is the generator framework which transforms the UML models into an implementation. It supports transformations into several technologies and it is possible to add new transformations. In general, it works with the class diagrams and based on the class stereotypes, it generates the source code. Moreover, it can be extended to work with other diagram types. A similar generator (made as an Eclipse extension) is openArchitectureWare [12], which is a general model-to-model transformation framework.

In [15], the sequence diagrams together with class diagram are used to generate fragments of code in a language similar to Java. The generation is based on the order of messages captured in the sequence diagram and the structure of the class diagram. There is also a proposed algorithm for checking consistency between these two types of diagrams.

Similarly in [5], Java code fragments are generated from the collaboration and class diagrams. The authors use enhanced collaboration diagrams in order to allow better management of variables in the generated code.

In [16], the use cases are automatically parsed and together with a domain model are used to produce a state transition machine, which reflects behaviour of the system. From the high level view, the used approach is very similar to our solution but they allow for processing only very restricted use cases and thus the approach is quite limited.

7. Conclusion and Future Work

The approach proposed in the paper allows for automated generation of executable code directly from a requirement specification written as use cases in a natural language. Also, we have developed a prototype, which generates JEE applications via the proposed approach.

Applications generated by our tool are immediately ready to be deployed and launched and they are suitable for testing the use cases (i.e. if the requirement specification is complete and well structured) and as a starting point for the development of the real implementation.

The proposed generator has several issues, which suit for further improvements. An important issue is connected with associations among the classes in the domain model. The current implementation correctly handles just one-to-one associations. The one-to-many or many-to-many associations result in the code to lists or arrays and therefore the determination of the arguments is more complex. We plan to solve association limitation by analysis of sentence to determine whether a method argument is the list or object itself. Also, we plan to add a categorization of verbs to allow better management of arguments of the procedures. We plan to employ some platform independent template framework which will enable to generate more configurable system implementation for several platforms. The planned output of the generator then would be a XML file which will be an input

for the employed framework. Finally, we plan to add generation of the data layer to applications.

The required structure of the use cases (based on recommendations in [3]) can be seen as another limitation but we already have an approach, which allows processing of use cases with almost any structure (see [4]) and we are incorporating it to the implementation.

Acknowledgements The authors would like to thank Vladimir Mencl, Jiri Adamek, and Pavel Parizek for valuable comments. This work was partially supported by the Czech Academy of Sciences project 1ET400300504.

References

[1] B. Anda and D. I. Sjober. Investigating the role of use cases in the construction of class diagrams. *Empirical Software Engineering*, Volume 10(3), Jul. 2005.

[2] AndroMDA. http://galaxy.andromda.org.

[3] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, Jan. 2000.

[4] J. Drazan and V. Mencl. Improved processing of textual use cases: Deriving behavior specifications. In *Proceedings of SOFSEM 2007, Harrachov, Czech Republic*, Jan. 2007.

[5] G. Engels, R. Huecking, S. Sauer, and A. Wagner. UML collaboration diagrams and their transformation to Java. In *Proceedings of UML '99*, Fort Collins, USA, Oct. 1999.

[6] M. Fiedler, J. Francu, V. Mencl, J. Ondrusek, and A. Plsek. Procasor environment: Interactive environment for requirement specification. http://dsrg.mff.cuni.cz/~mencl/procasor-env.

[7] JBoss application server. http://jboss.org.

[8] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall PTR, 2nd edition, 2001.

[9] V. Mencl. Deriving behavior specifications from textual use cases. In *Proceedings of WITSE '04, Linz, Austria*, Sep. 2004.

[10] Objecteering software. Objecteering 6. http://www.objecteering.com.

[11] OMG. Model driven architecture (MDA). OMG document ormsc/01-07-01, Jul. 2001.

[12] openArchitectureWare. http://www.openarchitectureware.org.

[13] F. Plasil and V. Mencl. Getting "whole picture" behavior in a use case model. In *Proceedings of IDPT*, Austin, Texas, USA, Dec. 2003.

[14] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, Volume 28(11), Nov. 2002.

[15] L. Quan, L. Zhiming, L. Xiaoshan, and H. Jifeng. Consistent code generation from UML models. UNU-IIST Rep. No. 319, The United Nations University, Apr. 2005.

[16] S. S. Somé. Supporting use cases based requirements engineering. *Information and Software Technology*, 48(11):43–58, 2006.

[17] Sun Microsystems. Core J2EE patterns: Best practices and design strategies. http://java.sun.com/blueprints/corej2eepatterns.