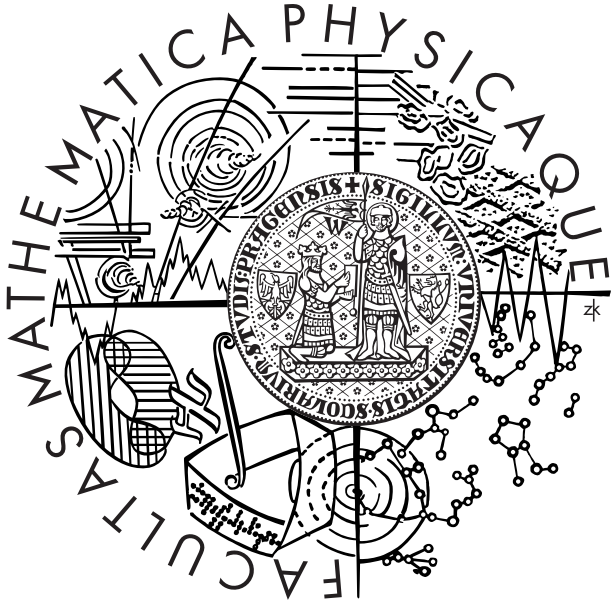
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Jan Franců

Generating implementation from system requirements

Department: **Department of Software Engineering**
Supervisor: **RNDr. Petr Hnětynka, Ph.D.**
Study program: **Computer science**

I would like to express my thanks to Vladimír Mencl for the ideas that he brought to this work and his precious time of tuition. I would like to thank my supervisor Petr Hnětynka for guiding me while I was working on this thesis and his reading the text and making valuable suggestions. I also wish to thank Jiří Ondrušek for his advice on the application structure.

I also thank my family for their support.

This is to confirm that I wrote this thesis on my own and that the references include all the sources of information that I explored.

I agree that this thesis can be used for academic purposes.

Prague, 10th August 2007 Jan Franců

Title: Generating implementation from system requirements

Author: Jan Franců

Department: Department of Software Engineering

Supervisor: RNDr. Petr Hnětynka, Ph.D.

Supervisor's e-mail address: `Petr.Hnetynka@mff.cuni.cz`

Abstract: Designing a software system starts with writing the system requirements. Typically, these requirements are expressed in UML and contain use cases and domain model. A use case is a sequence of tasks which have to be done to achieve one of the system's goals. These tasks are written in natural language (English). The domain model (typically captured in a class diagram) describes used objects and their relations.

The goal of the thesis is to analyze, whether the system requirements are sufficient for generating an implementation model, which will manage communication between the system's components. The generated model might be used for future development and can be used for testing the users interactions. A prototype implementation of the generator is expected.

Keywords: procase, use case, implementation model, domain model

Název práce: Generování implementace ze systémových požadavků

Autor: Jan Franců

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Petr Hnětynka, Ph.D.

E-mail vedoucího: `Petr.Hnetynka@mff.cuni.cz`

Abstrakt: Vytváření softwarového systému začíná sepsáním systémových požadavků. Obyčejně jsou tyto požadavky zapsány pomocí UML a obsahují mimo jiné use casy a domain model. Use case je sekvence úloh, které je potřeba udělat k dosáhnutí jednoho z cílů systému. Úloha je zapsána v přirozeném jazyce (angličtina). Domain model, zpravidla zachycen jako class diagram, popisuje použité objekty a jejich relace.

Cílem této práce je zjistit, zda specifikace systémových požadavků jsou postačující pro generovaní implementačního modelu, který bude řídit komunikaci mezi systémovými komponentami. Model by měl být použitelný pro další vývoj a pro testování uživatelského rozhraní. Bude předložen prototyp programu generujícího implementační model.

Klíčová slova: procase, use case, implementační model, domain model

Contents

Chapter 1

Introduction

1.1 Motivation

The software developers can take several different approaches to developing software. One of them is the Unified Modeling Language [13], which has become the industry standard at least for large and semi-large enterprise applications. Development with UML (so called UML pattern) is based on modeling the developed system in several levels of abstraction. The models help developers to work at a certain level of abstraction, reflect specific aspects and get the big picture of the developing project.

The development according to the UML pattern starts with the following steps:

1. Define goals of the developing project.
2. Inspect already implemented solutions and their advantages and disadvantages.
3. Describe main characteristics of the system requirements.
4. Create project use cases and domain model.

The documents created in the fourth step are employed in this work. The software developers are describing the system behavior with documents called use cases, where the Use case is a description of a task performed by the designed system. The designed system consists of entities which communicate with each other. The entity must proceed certain steps while achieving the task described in the use case. These are called use case steps and are captured in natural language sentences. The Use case consists of a use case step sequence. Second document created in the fourth step is the Domain

model which is a description of objects used in the project (and in the use cases) and their attributes and relations.

Today the software development is usually an iterative process, where the system requirements are revisited each iteration. The weakness of this process is the time cost, thus it would be helpful to accelerate an iteration or to enable the developers to skip an iteration. This can be done if the developers can generate an implementation model of the designed system just after they have created the system requirements (Use case and Domain model).

The project use cases and domain model contain all the most important information about the designed system. These documents can be processed with the Procasor program [18] to deliver a code-like record of the system (Procase) which with some additional information can be used for a generation of the implementation model.

The process, proposed by this thesis, enables software developers to take an advantage of the carefully written system requirements to accelerate the initial iteration and it can provide some feedback for the project's analysts to see whether some elements of the system requirements are missing. A project with its source files is generated by the proposed generator program and then an implementation model application is built from this project. The generated project can be used for further development and the implementation model can be used for testing purposes.

1.2 Goals

The goal of this thesis is to explore the possibilities of using the analyzed use cases and the domain model for the generation of the implementation model as well as to decide whether these documents contain sufficient information and evaluate qualities of the generated implementation model. The relations between the use cases and the domain model are analyzed for a correspondence with method arguments generated in the implementation model. The intended logic structure of the implementation model is derived as well as construction of generation of the reasonable implementation model project source files which would be possible to use as the first stage for the future implementation.

The proposed approach is supported by a proof of concept prototype implementation of the generator program.

1.3 Structure of the Thesis

In Chapter 2, an overview of the background of the thesis is provided. The UML software development pattern is introduced together with the terms Use case and Domain model. The next introduced topic is Protocol use case referred to as Procase together with more advanced aspects added into the Procase language.

Chapter 3 contains the thesis solution, Section 3.1 presents a brief description of the whole proposed process. Sections 3.2, 3.3 and 3.4 focus on each main step of the proposed process.

Chapter 4 records the proposed process application on a case study project and a evaluation of the generated application.

The related work and a discussion over the future aims and the thesis conclusion are presented in Chapter 5.

Chapter 2

Background

The basic knowledge of software development according to the UML pattern [13] is presented in this Chapter, with focus on the documents created during collection of requirement (Use cases and Domain Model). Another topic introduced is a Procase language which describes the process captured in the Use case more elegantly.

2.1 UML

The *Unified modeling language* [13] is a standardized specification language for the software development. The UML pattern is based on modeling the developing system at several levels of abstraction in several models. Each model, represented as a set of documents, clarify the abstraction and/or capture different aspects of the modeled system. The UML pattern also a standardizes when and which documents should be created. This process is used by the developers to ensure that the system will meet all the requirements. UML also enables the reuseablity of the models and makes reuseablity of the implemented modules easier. Today UML is an industry standard and is used around the world. Not everyone follows the standard consistently and therefore there are many variations but the concept of the UML pattern applies. Beside the text documents UML includes graphical notations which are characteristic of the UML pattern. The graphical notations can be serialized in a standardized XML (called XMI), which enables computer processing and maintaining.

This thesis works with a set of documents written when the requirement specification of the developing project is gathered. The Use cases and the Domain model are introduced in the following sections.

2.2 Use case

Use case in context of UML is a description of a process where a set of entities cooperates to achieve an use case goal. Communications among entities and significant actions are essential to this process. The entities in the use case can refer to anything taking part in the process, nevertheless usually they refer to the system, users or components. Every use case has one entity called *SuD* (*system under discussion*) from the perspective of which the whole use case is written. Another entity with which the *SuD* mainly communicates is called *primary actor* (abbreviation *PA*), other involved entities are called *supporting actors*.

This description can be structured as a document called the Use case in UML manner. For the purpose of this work, it is appropriate to know that a use case includes a *header*, a *main scenario*, *extensions* and *sub-variations*.

The header contains a name of the use case (should refer to the goal of the use case), *SuD* entity, primary actor and supporting actors and a *use case abbreviation*. The main scenario contains a list of actions that are performed when achieving the goal of the use case. The main scenario can also be referred to as the *success scenario*.

Each action has a *step label* indicating its position in the sequence of the process. Each action in the main scenario is described in a natural language sentence. An action can be extended with a *branch action*, which reflects possible diversions from the main scenario. The Use case has two types of branch actions: the type of *extension* where actions are performed in addition to the extended action and the type *sub-variations* where actions are performed instead of the extended action. The first action of the branch actions is the *conditional label* which describes the case when the branch actions are performed.

Use cases can be written in many ways, but the previous paragraphs refer to Cockburn recommendations [6] of writing effective use cases. Use cases can be written in any language, but this thesis will concentrate on the use cases written in English because there is a tool (see 2.5) transforming English written use cases into procases (see 2.4) which is essential for this work. Consistent adherence to the Cockburn suggestions enables the smooth use case transformation with the intended result procase. The following example demonstrates a use case written in the correct way. The use case describes communication between entities of the Seller and the Marketplace Information System (as SuD).

Use Case Example:

Use Case: Seller submits offer
SuD: Marketplace Information System
Primary Actor: Seller
Abbreviation: UC1

Main success scenario specification:

1 Seller submits item description.
2 System validates the description.
3 Seller adjusts/enters price and enters contact and billing information.
4 System validates the seller's contact information.
5 System verifies the seller's history.
6 System validates the whole offer with Trade Commission.
7 System lists the offer in published offers.
8 System responds with a uniquely identified authorization number.

Extensions:

2a Item not valid.
2a1 Use case aborts.
5a Seller's history inappropriate.
5a1 Use case aborts.

2.3 Domain model

Domain model is a document created during the collection of the system requirements. The Domain model describes objects used in the project; typically captured in a class diagram [13]. The class diagram capturing domain model consists of three types of logical objects: *Conceptual classes*, *attributes* of a conceptual class and *relations* between conceptual classes. The Conceptual classes refer to objects used in the Use cases. They represent objects of interest in the modeled process. The attributes refer to the features of the represented objects. The relations between the conceptual classes reflect references or composition between the objects of interest. The attributes can be completed with types, which must be simple like string, number of date not

a object. Figure 2.1 demonstrates the graphical notation of two conceptual classes and their attributes and a relation between them.

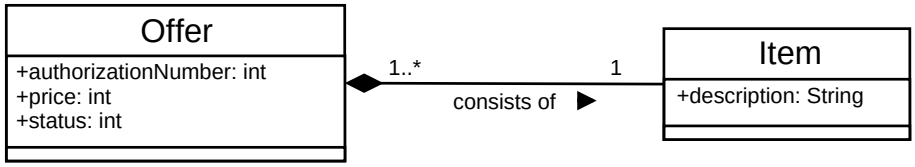


Figure 2.1: The example of the Domain model.

2.4 Procase

A *protocol use case* (*Procase*) [16] is based on *Behavior Protocols* [17]. It is a regular expression-like notation, which describes an entity behavior. The Procase can describe also a whole system behavior [15]. The *traces* generated by the expression represent all the possible valid sequences of the entity actions described in the use case.

Each entity action has its token which describes the action. The action label notation starts with a action type sign followed by the entity acronym (entity which is triggering the action). Then follows a dot for separation and the action token. The entity acronym and a dot is left out for the internal actions. There is also an empty action *NULL*.

Possible types of an action are:

?	request receive
!	request send
#	internal
%	special

The *?SL.submitItemDescription* is an example of the action label where *?* means that the action is received by the SuD entity and the action is triggered by the entity with the *SL* acronym (Seller) and the action token is *submitItemDescription*.

The Procase notation has the same basic operation as the regular expression. With this operations the Procase notation is able to describe the possible behavior of the designed entity. As an atomic piece of the Procase, the action labels are combined together with a Procase operations to obtain the intended entity behavior.

Procase operations are:

operator	name	meaning
+	alternative	$a + b$ means a or b is performed
;	sequencing	$a; b$ means b is performed just after a
*	iteration	$a*$ means that a is performed zero to finite number of times
()	supporting brackets	changes the priority of operators
{}	procedure calls brackets	encapsulates procedure call actions

The Procase operators have priority in the following order: iteration (*), sequencing (;) and alternative (+) from the highest to the lowest priority. The supporting brackets can be used to control the order in which the operations are evaluated.

In some cases, actions are mentioned without the type sign, that denotes the actions without importance to the context. Usually these actions are internal actions. The actions can be mentioned with the request receive type sign and without an entity acronym which denotes that the triggering entity is not important to the context.

The alternative operator together with its arguments is called the *branch action* and each argument is called the *branch*. The iteration operator with its argument is called the *loop action*.

2.4.1 Procedure calls

The procase can be completed with a *procedure call* brackets where the procedure call encapsulates actions which are done as a one call on the SuD entity. Steps of the success scenario can be divided into several procedure calls. The procedure call is represented as a pair of procedure brackets (curly brackets) with a *trigger request receive action* placed in front. Inside the pair of procedure brackets there are actions which belong to the procedure call (mainly internal actions). Another request receive action cannot be located inside the procedure call.

Example of a simple procase completed with the procedure call:

$$?a\ \{\#b; !c; \#d; \#e\}$$

The procedure call shown above is triggered by the a action and the procedure call contains a sequence of actions b, c, d and e, where c is the request send action and other actions are the internal actions. The SuD entity performs the sequence of actions when receives action a as one task.

2.4.2 Use case expressions

Use case expressions are used to draw the whole picture [15] of an entity or the whole project. The Use case expressions are used for determining the possible behavior of an entity or of the project. The Use case expressions language has a regular expression-like notation and uses use case abbreviations as the atomic pieces. The Use case expressions language has following operators:

operator	name	meaning
;	sequencing	$UC\_A; UC\_B$ means use case UC_B is performed just after use case UC_A
+	alternative	$UC\_A + UC\_B$ means use case UC_A or use case UC_B is performed
*	repetition	UC_A* means that use case UC_A is performed zero to finite number of times
||	parallel run	$UC\_A \,\|\|\, UC\_B$ means that both use cases interleaving

The operators have the following priority: repetition (*), sequencing (;), alternative (+) and parallel run (||) from the highest to the lowest priority. The supporting brackets can be used to control the order in which the operations are evaluated.

2.5 Procasor program

The *Procasor* Environment program [18] is a tool for writing project use cases and for an interactive generation of procases. The program is based on the transformation invented by Vladimír Mencl and described in his PhD thesis [12]. This transformation has been improved in the Jaroslav Dražan Master thesis [7] supervisored by V.Mencl; allowing to derive a procase from more complex sentences.

The Procasor program enables software developers to write the project use cases inside this tool or use cases can be loaded from the text files (with a certain form). Then the use cases are transformed into the corresponding procases. Alongside with this transformation, several information can be obtained like the type of action, representative object, subject etc. It is possible to save the whole project (analyzed use cases with the additional information and procases) into the XML file or export to the UML state diagrams. More about the Procasor program can be found in the project documentation, see [18].

2.6 Procase enhancement

To enable an automatic completing the Procase with the procedure calls it is necessary to enhance the Procase language with two simple constructions; Conditional events and Aborts are formulated and elaborated as a part of this work.

2.6.1 Conditional events

It is necessary to add a request receive action inside a branch of a branch action many times during writing use cases. Then a problem of having well formed procedure calls can occur during adding the procedure call brackets. The following example demonstrates a procase which cannot be completed with the procedure calls brackets:

$$?a; b; (c; ?d; e + g; h; i); j$$

Example above cannot be completed with the procedure call brackets because actions e and j should be enclosed in two procedure call brackets.

Hence some kind of transformation is needed to obtain a procase where it is possible to mark the procedure brackets. The transformation must preserve the procase traces. The *Conditional events* construction allows this transformation. Enhanced syntax of Procase language should preserve easy readability of the notation, trace nature and regularity.

The new special action is added to Procase syntax. The action starts with "$" symbol followed by a declaration of the variable, written in capital letters. The value of this variable from this point is *true* and it is local for the use case. The action with "$\sim$" in front of the variable name can be used to set the value of the variable to *false*.

Also the alternative operator is changed. The variable name with a colon can be placed in front of the particular branch of the alternative operator. Then only the traces which contains the action with declaration of this variable and when the value is *true* continue with this marked branch. When the value is *false*, the other traces continue with the unmarked branches as it was the former alternative operator. All variables are initialized as *false* at the beginning of each use case.

It is important to preserve regularity of Procase language enhanced with this construction. This language (Procase) is regular if and only if it is acceptable with a finite automaton. The amount of variables in Procase is finite and therefore the Cartesian product of the set of former states and all the variables as true and as false gains finite automaton too.

Example of a enhanced procase syntax:

$$a; b; (c; \$D; e + f); g; (D : r; s; t + u; v; w); x; y; z$$

2.6.2 Aborts

For the purpose to distinguish the traces with failure ending from others with success ending, it is necessary to enhance the Procase syntax little more. The new special actions *%ABORT* and *%USE_CASE_TERMINATE* are added to the Procase syntax. The traces where are located these special actions terminate after the first location of this action. The *%ABORT* action denotes the failure of the use case and *%USE_CASE_TERMINATE* denotes the success ending. It is not necessary to mark all success ending with the *%USE_CASE_TERMINATE*; only the failure ending is necessary to mark with the *%ABORT* action. The enhanced syntax of Procase should also preserve easy readability of the notation, trace nature and regularity.

The Procase language enhanced with these special actions must preserve the regularity. This language (Procase) is regular if and only if it is acceptable

with a finite automaton. These special actions can be represented as two new special end states, which are added to the former finite automaton. This new automaton is still finite, therefore this enhanced Procase is still regular.

2.7 Additional entity type

The entity type is an important information for generating the implementation model. There are two types of entities which cooperate inside the client-server application: users (humans - seller, buyer, client etc.) and computer systems, system components or modules. The entity implementation which represents the user has typically a user interface. The system entity implementation has only a program interfaces for communication. In the client-server type application the user interface is realized by web pages and thus entities which need to have the user interface are called web fronted entities (WFE) in this work. The other entities are called non web fronted entities ($nonWFE$).

Chapter 3

Generating process

3.1 Overview of proposed process

Proposed process which enables to accelerate the software development requires certain work flow. Software developers, analysts or programmers should write use cases in the Procasor program, see 2.5, according to the Larman [11] suggestions and with care to produced procases and used action tokens. As complementary document of the system requirements, the domain model should be created; contains the conceptual objects with their relations and attributes.

At this point, the proposed generator should be started. It produces the first draft of the implementation model. The model can be analyzed and tested and the results made with this analysis can be used while rewriting the use cases or the domain model. After then the proposed generator should be started again to get more observations. Once the implementation model is satisfied, then the implementation model project can be used for further development. Figure 3.1 illustrates the whole proposed process.

The proposed generator program proceeds in three steps listed below which are described in following sections.

- The first step (described in Section 3.2) is to arrange the procase into a form which more tightly refers to the implementation, but still follows the Procase syntax.
- The second step (described in Section 3.3) is to analyze connections between use cases and the domain model and to create action arguments. The action arguments are generated as method arguments in the implementation model.

- The next step (described in Section 3.4) is to generate a reasonable implementation model project which will be possible to use as a first stage for the future development. On the application, built from the implementation model, should be possible to evaluate the use cases and test for missing elements. The generation process includes deriving intended internal logic structure and creating the application framework.

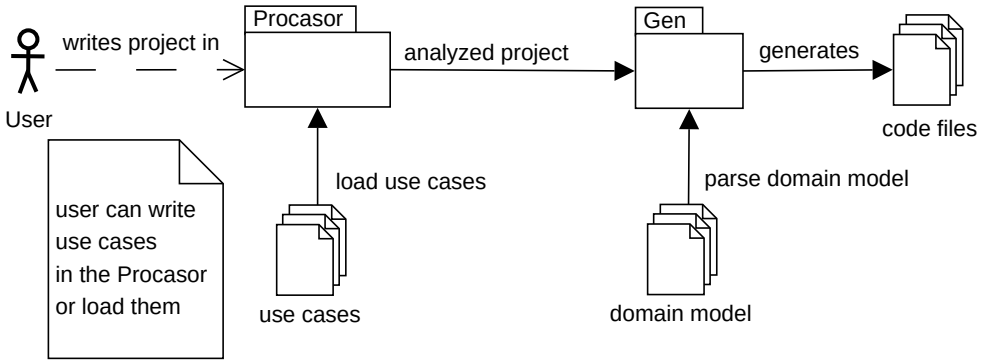


Figure 3.1: The proposed process overview.

Larman [11] highlights two common misunderstanding during the requirement collection. The first is to think about the system requirements and design as backward unchangeable statements. The second is to heavily focus on the design before implementing. Thus, the proposed process generator is used with the uncompleted and unfinished inputs (Use cases and Domain model), therefore it can obviously generate a completely misleading implementation model.

3.2 Preprocessing

The Preprocessing transforms Procase into the form which is acceptable for further processing.

The generating process use as input procases completed with the procedures calls. The procedure calls refer to methods of generated objects. Along with this process of assigning actions to the procedures, the Preprocessing corrects flaws of the input procase. The Preprocessing starts with *Procase transformations*, where these flaws are corrected and then the *Procedure marking* follows.

Project procases and procases after Preprocessing can be found in the generated project inside the *procase* directory. There are files with suffix

"_app" where is the use case procase after the Preprocessing and there are files with no suffix with the original use case procases.

3.2.1 Procase transformations

As created with the Procasor program (see 2.5), the output procase can have redundant actions. The branch transformation corrects this flaw and consists of two steps. First, it optimizes all branches of the procase (Branch joining) and then it corrects the initial branch (Initial branch joining).

Branch joining

The flaws of Procase generated by the Procasor program [18] reside in duplicate prefix actions inside the branches. The Branch transformations joins these branches together and reduces the duplicities. More detailed, the branches with a common prefix are transformed into a sequence of actions. The sequence of actions is constructed using the common prefix actions of the original branches followed by branch action with the rests of the original branches.

Example of a procase with branches before and after the Branch transformations:

$$a; b; c + a; b; d + a; e; f + g; h; j + g; k; l$$

$$a; (b; (c + d) + e; f) + g; (h; j + k; l)$$

Initial branch joining

The common flaw of Procase generated by the Procasor program (see 2.5) is that an aborting trace is separated from a procase as a first branch of a branch action and a second branch is the remaining procase.

The initial branch joining transforms a use case which consists of the only one branch action. It joins the branches which end with the abort action together with the branches which do not. A branch is joined with another branch only if there is a common prefix of the actions. It is similar to the Branch transformations but it differs in that the aborting branch can be joined into another branch or even in the branch of the branch action located in as the first action in the former branch. The transformation does not

change the traces of the procase, because the joined branch ends with the abort action.

Example of a procase before and after initial branch joining where the first branch *a;b;%ABORT* is joined into the second branch which starts with the branch action.

$$a; b; \%ABORT + (a; b + c; d; e); f; g$$

$$(a; b; (\%ABORT + NULL) + c; d; e); f; g$$

3.2.2 Procedure marking

Procase, derived from the use case by the Procasor program (see 2.5), can be completed with the procedure calls (see 2.4.1). In basic, the procedure call starts with a request receives action and ends with the end of the procase or before another request receive action. There cannot be another request receive action within a procedure call.

Example of procase before and after procedure marking:

$$?a; b; c; d; ?e; f$$

$$?a\{b; c; d\}; ?e\{f\}$$

If it is not given in the preconditions, then the Use case is sequence of the request-response actions from the triggering entity point of view. Therefore, every Procase should start with a request receive action. The procases which do not start with the request receive action are handled in a different way. The actions located before the first request receive action are assigned to a *INIT* procedure call which is called before following procedure call.

If the request receive action is located in a branch or in a loop (even located in more branches) the transformation becomes more complex. This transformation is necessary to obtain a well formed procedure calls brackets – to avoid request receive actions be located inside the procedure calls. The Procase Conditional events construction, see 2.6, is employed for the transformation. The special cases which are resolved in this thesis are transformed with *Branch transformation* and *Loop transformation* and some cases left unresolved.

Branch transformation

If the request receive action is located in a branch (even that branch is located in more branches or loops) then the straightforward marking process fails. In this case the Branch transformation is applied. It places a declaration action of a conditional event variable (see 2.6) instead of the request receive action which is located inside the branch action. The rest of the actions in this branch are moved to the new procedure. The new procedure is placed inside the new branch action as one branch and this branch is marked with the same variable. The *NULL* action is created as the second branch. The newly created branch action is moved after the top level branch action. This part of the transformation is captured in picture 3.2.

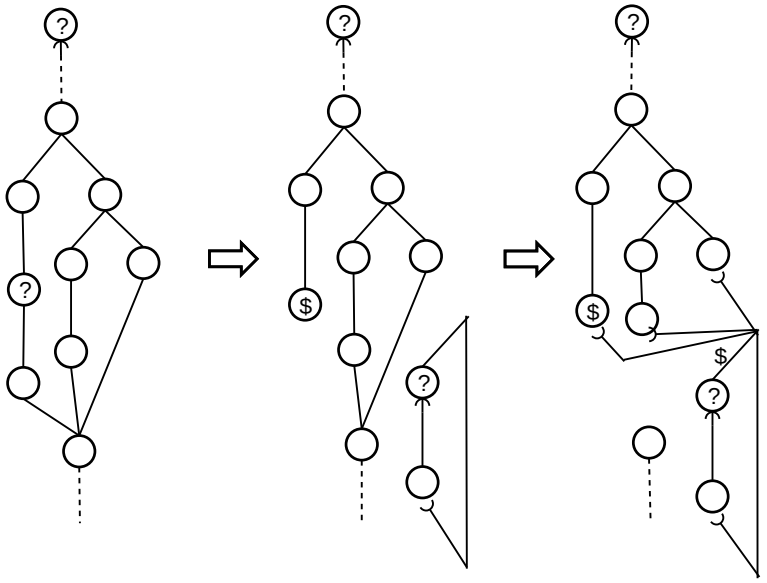


Figure 3.2: First part of the Branch transformation.

The actions which were after the top level branch action (until next request receive action) are appended to the branches of the top level branch action, except for the one branch where is the variable declaration. These actions are also placed at the end of the new procedure. If the variable declaration is located inside more branch actions then for each branch action is necessary to append actions which follows the branch action to each of the branches except for the one branch where is the variable declaration. This redistribution guarantees that all the traces which have the declaration of this conditional event variable have this declaration as the last action which

generates the top level branch action. This transformation process a procase in backward direction. The second part of transformation is captured in picture 3.3.

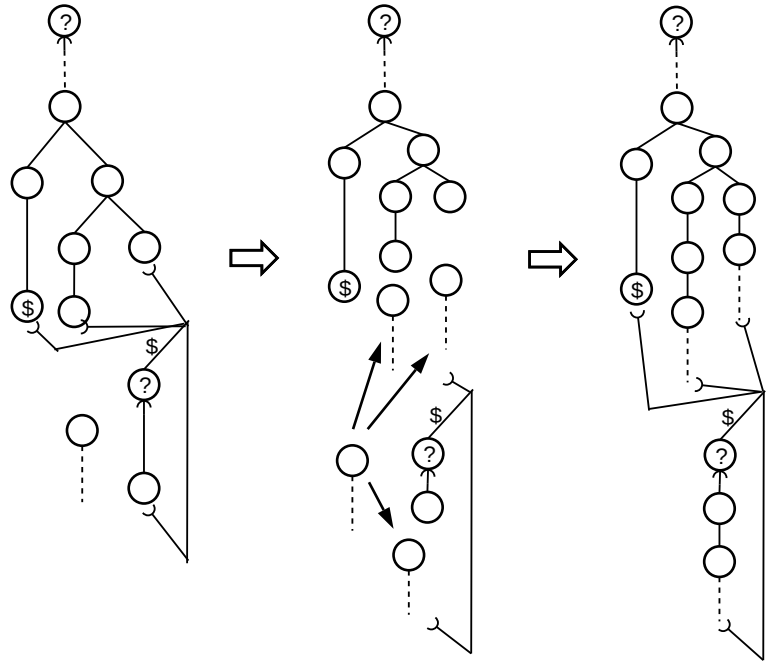


Figure 3.3: Second part of the Branch transformation.

Example of a branch before and after transformation where a request receive is located inside a branch action. The branch action was altered and a new procedure call was created:

$$?a; b; (c; ?d; e + f; g + h); i; j$$

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

Loop transformation

This transformation is similar to the Branch transformation. The request receive action located inside a loop action is set as the trigger action of a new procedure. And a new conditional event variable declaration replaces this action in its former position. Actions which follow this declaration are placed within the new procedure. A new branch action is placed after these actions within the new procedure. The actions previously located inside the loop before the conditional event variable declaration are placed as the first branch. The actions following after this loop are placed as the second branch with unsetting conditional event label variable action at the end. These transformations are captured in picture 3.4.

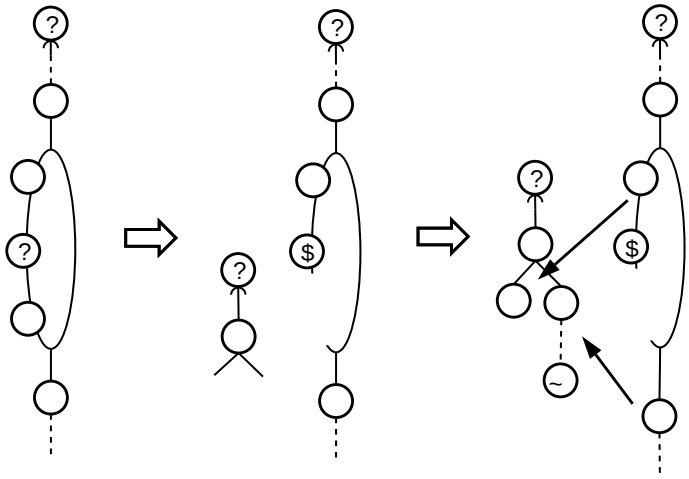


Figure 3.4: First part of the Loop transformation.

The new procedure is placed inside a new branch action as branch marked with the same label variable which is used inside the loop. The *NULL* action is created as the second branch. And finally this branch is placed inside the newly created loop.

The part of the actions which were inside the loop before the variable declaration (previous location of the request receive action) is encapsulated inside the new branch action as a one branch. All actions which were formerly after the loop are placed as the second branch. These actions are the same actions as the second branch inside the new procedure, just without unsetting label of the conditional event variable. The Loop transformation process a procase in backward direction. The second part of transformation is captured in picture 3.5.

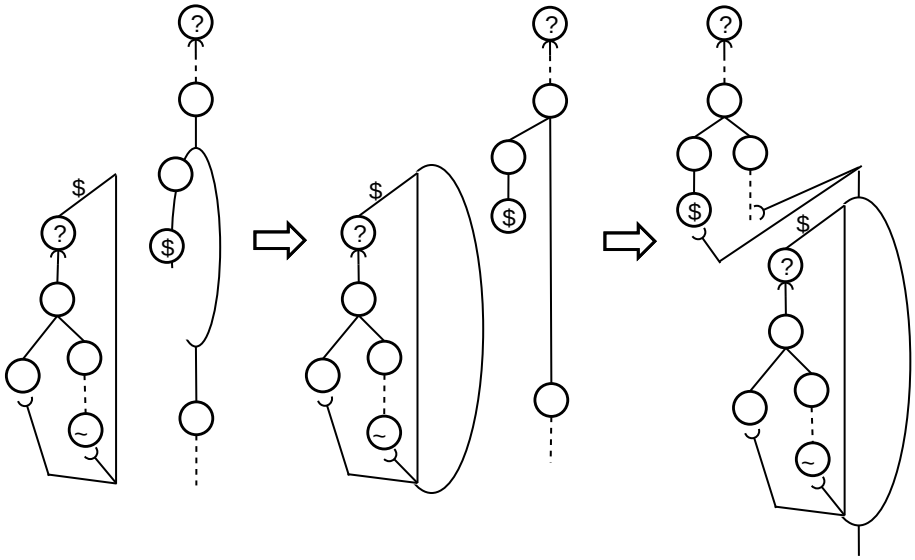


Figure 3.5: Second part of the Loop transformation.

Example of a procase before and after the Loop transformation where a request receive action is located inside a loop action. The transformation removed the loop action and it placed a new branch action. A new procedure call within a branch and a loop action was appended:

$$?a; b; (c; ?d; e) * f$$

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

Another example of a more complex procase (with a branch inside) before and after the Loop transformation:

$$?a; b; (c; d; (e; f + g; ?h; i); j) * k$$

$$?a\ \{b; ((\sim H : c; d; (e; f; j + g; \$H) + NULL)*) + k)\};$$
$$(H : ?h\ \{i; j; (((c; d; (e; f; j + g) + NULL)*) + k; \sim H)\} + NULL)*$$

Unresolved cases

If a request receive action is located inside two or more loops or if it is inside a loop and the loop inside a branch action then these cases are marked as not processable and this procases are skipped during the Preprocessing. Larman [11] suggests that maximum complexity of Use case's extensions/sub-variations is a branch inside a branch. This level of complexity is handled by the proposed the Preprocessing process. More complex constructions are badly readable while readability is the main feature of Use case. Therefore use cases which were skipped during the Preprocessing (Procedure marking step part) should be rewritten, divided or otherwise changed.

3.3 Determining arguments

The aim of this process is to determine arguments of an action according to the project domain model, use case step sentence and available variables in the procedure. These action arguments are generated as methods arguments in the implementation model.

3.3.1 Correspondence between domain model and noun phrases in use cases

Noun phrases in project use cases can be used for determining the conceptual classes for the project domain model (described in [11]). This technique derives the noun phrases from the use cases and collects candidates for the conceptual classes or for the conceptual classes attributes. Some of these noun phrases can mean the same thing because of the imprecision of the natural language; some of them should not be mentioned in the domain model. However, in many cases the noun phrases in the use cases refer to a certain conceptual classes or attributes in the domain model. This fact is used when the action arguments are determined from the use case step.

The correspondence between a Class model and use cases is investigated in the work [2]. The Class model is captured in the same diagram notation as the Domain model, but it is more detailed and contains class methods. The work result confirms that the described technique for deriving the class model from the use cases is the most effective among other techniques and therefore there are correspondences between noun phrases in use cases and the domain model which is kind of ancestor for class model.

3.3.2 Overview of determining arguments

The general process of determining arguments is as follows. The list of potential words is extracted from the use case sentence. The list is matched against the words used in the domain model and the target types are determined. In case that this action is called on WFE entity (see 2.7), then all determined types are used as action arguments. Else the available variables which have matching types are extracted from the use case procedure calls and these variables are used as arguments of the action. In following paragraphs is described each step in more detail.

Extracting words from sentence

From the use case step sentence, it is necessary to extract the words which may refer to the data used/filled in/changed in the use case step. In addition to the use case step sentence this process uses the information derived from the sentence by the Procasor program, see 2.5. The Procasor program employs linguistic tools to acquire information about an action expressed by the sentence. Each word in the sentence is analyzed for its constituent of the sentence. More information about the linguistic analysis can be found in the documentation of the Procasor program [18]. The following words are extracted from a use case step:

- nouns – all nouns potentially refer to arguments
- representative object – the Procasor program (see 2.5) extracts nouns from the use case sentence and matches them against the conceptual objects list (more in [18]); the Procasor marks the matched phrase as the representative object
- verbs – the relations between the conceptual classes in the domain model are usually expressed as a verb phrase
- entity names – entity name which is triggering the action is excluded
- words used in action token – usually redundant information, but it could bring some new potential words; in case that the token was created by a Procasor program user, see [18]

Matching extracted words against domain model keywords

The words extracted from the use case sentence are matched against keywords in the domain model. The keywords of the domain model are names of the conceptual classes, names of the attributes and names of the relations between the conceptual classes. The extracted words are matched against the keywords in a simple way by an equals ignore case method. If there are more keywords matched for one extracted word, then a conceptual class have higher priority than a conceptual class attribute and than conceptual class relation. In case that a conceptual class attribute is matched then the conceptual class name is excluded, because the conceptual class name is usually used in a use case sentence only to identify the conceptual class attribute.

Matching extracted types against available variables

The available variables are extracted from the procedure call where the action is located. The variables are inputted as procedure call arguments or are inputted from the WFE entity (see 2.7). Also the variables are extracted from previous procedures (these variables are marked as session variables, see 3.4.8). Previous procedures which are in a branch beside the *NULL* action are excluded from processing, because they may not be called before the processed one.

The available variables are matched (by type and name) against the requested types where variables previously assigned have the priority. In case that an available variable refers to the domain model class then the requested type can be matched against that class' attributes types.

3.4 Application generation

The Java Platform, Enterprise Edition [21] with the Java Server Faces [22] has been chosen for the implementation model, because it is the commonly used framework for large enterprise applications today. The generation is focused on the internal business structure and the testing purpose web frontend.

The generation process does not produce a product application (in this case an .ear file), but it generates a java project file structure with an Ant build file [19]. Finally, the Ant build generates the product application which can be deployed to an application server.

3.4.1 Generic structure of Java EE application

The structure of the generated application is created according to the chosen technologies. The application is structured into a tiers. The web frontend is created with JSF technology [22] and has two tiers - JSF pages and Backing beans. The next tier is a Business delegator tier which encapsulate an EJB tier according to the Business delegator design pattern [20]. The next tier is the Enterprise java bean (EJB) tier where the EJBs contains the use case internal logic. The following tier is a Manager tier which contains an internal implementation of the entity actions. The actions implementation is for the testing purpose implemented as a procase trace logger.

The intended following tiers are a Data access objects which encapsulate the database access and a Data persistence tier usually represented by a database. These two tiers are not generated and are marked with gray color in the design overview figure 3.6.

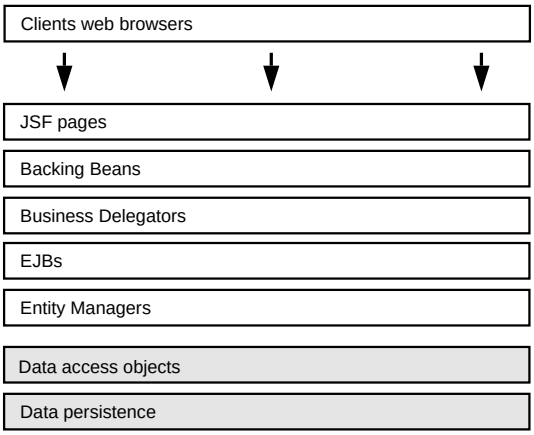


Figure 3.6: The structure of the generated application.

The following abbreviations are used in the following paragraphs and figures:

BB	backing bean
BD	business delegator
EJB	enterprise java bean
MGR	entity manager

The generated file system project structure has the common layout of the J2EE application project. The *src* directory contains the java source files where the most important are located in *ejb*, *mgr* and *core* packages. The *ejb* package contains the enterprise java beans (and their interfaces Local and LocalHome) which contains the intended internal logic structure of use cases (sequence of actions). The *mgr* package contain an implementation of the entity managers with action methods (implementation of actions). The *core* package contains an implementation of the conceptual classes used in the project domain model. The whole file system structure is captured in figure 3.7.

Navigation between web pages determine the order of the procedure calls and it is a part of the use case internal logic structure. The navigation interconnects the procedure calls and it determine also the order of the actions inside the procedure call for the use cases with the WFE SuD. The navigation is described in Section 3.4.7.

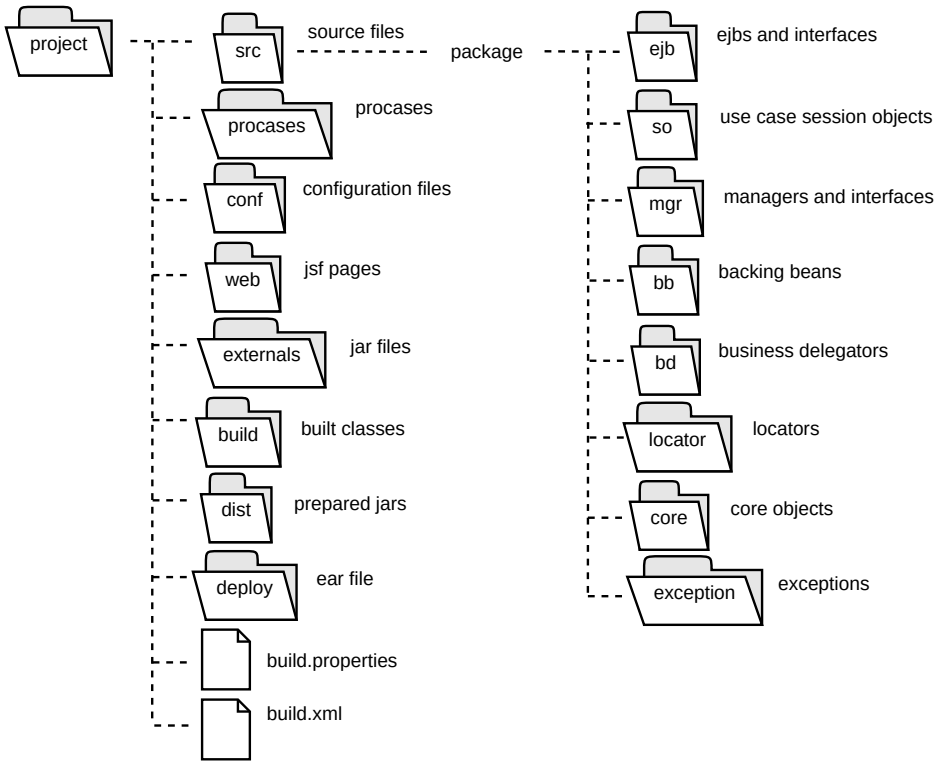


Figure 3.7: The generated file system structure.

Generation according to type of entity

Triggering action pages are generated for every use case which has the triggering WFE actor. A Backing Bean which is the backend for the generated pages and a use case BD and a use case EJB are also generated. The MGRs are generated alongside. The elements generated for the use case with the nonWFE SuD are captured in figure 3.8.

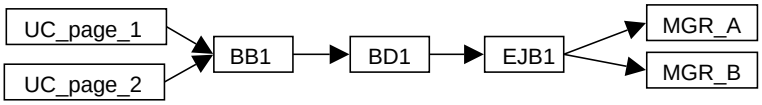


Figure 3.8: The elements generated for the use case with the nonWFE SuD.

Figure above shows two generated trigger action pages which are bound to the use case BB. The BB delegates calls to the use case BD which encapsulates the use case EJB. Each call in the EJB is delegated to the particular MGR.

The generation is more complicated for the use cases with the WFE SuD entity. The same classes (BB, BD and EJB) and the same pages as for other use cases with nonWFE SuD are generated. The internal procedure call action pages, backing beans and business delegators and EJBs are generated alongside. All these additional classes and pages are generated with a name suffix "X" (or in same cases the suffix starts with "X"), which denotes that it corresponds to the procedure call internal action. This separation enables the future development of the generated project. The elements generated for the use case with the WFE SuD entity are captured in figure 3.9.

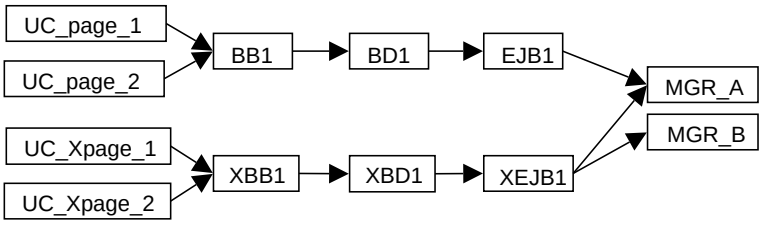


Figure 3.9: The elements generated for the use case with the WFE SuD.

Picture above shows four generated pages; two are bound to the use case BB and another two are bound to the use case XBB. The BB delegates calls to the use case BD which encapsulates the use case EJB; same with the X use case. Each call in the EJB and in the XEJB is delegated to the particular MGR. The first two pages refer to the triggering actions and X pages refers to a procedure call internal actions.

Every procedure call is one of the four type and according to this type the certain source files and/or pages are generated. The types are described in following table:

SuD - PA/supporting Actor	WFE	nonWFE
WFE	2 - 1	2 - 4
nonWFE	3 - 1	3 - 4

Table 3.1: The number combination refers to the type of the procedure call.

The generated elements are identified in the following paragraphs according to the procedure call type.

1. A procedure call trigger action page is generated for each procedure call with the triggering WFE entity. A method created from the trigger action is generated inside the triggering entity use case BB. The method body contains delegation to the use case EJB via the use case BD.

2. A method in the use case BD is generated for each procedure call with the WFE SuD entity. The BD method body contains delegation to the use case EJB where a navigation string for X pages is returned. Internal procedure call actions are generated in a similar way to the request receive action with the WFE PA. The X page and a method inside the X BB are generated for the internal procedure call action. The method inside the X BB and a method inside the X BD and a simple delegation method inside the X EJB are also generated.

3. A procedure body with the procedure actions inside is generated inside the use case EJB.

4. A action implementation method is generated in the actor MGR. The action method body contains the delegation to the use case EJB via the use case BD.

3.4.2 JSF pages

The JSF pages are generated for a testing purpose and are created as follows. Every WFE entity has a page for each of the action triggered on/by the entity. The PA WFE entity has a page for every triggering action and the SuD WFE has also a page for every procedure call internal action. Each page displays status information – the acting entity name and the current processing action. If the action has arguments then the page has an input fields for these arguments. For each argument is generated label with its name and a input field.

If the page belongs to the PA WFE action it has a button which enables to continue to the next page.

If the page belongs to the SuD WFE action it contains buttons which reflects the possibilities of the traces. For sequence of the actions, the page contains the "continue" button. If the next action is a branch action then the page contains for each branch a button. The main scenario branch has button with the caption "continue" and the other branch buttons has the

caption with the branch condition label action token. If the next action is a loop action then the page has the button to enter the loop and the next button to skip the loop.

The procase of the use case and the use case are located on the generated page. The current action label is highlighted in the procase and the current use case step is highlighted in the use case.

The index page for this application is also generated. In the index page it is possible to choose an entity and continue to the entity page. On the entity page is possible to choose a use case, where this entity is the triggering entity or it is possible to go back to the index page. An error page and abort page are also generated. From the index page a setting page is also accessible. Properties for enabling and disabling the branches and to determine the number of the iteration of the loop action can be set on the setting page. This setting can be applied for the use cases with the nonWFE SuD entity and also for branch and loop actions which are the first actions in the procedure call in the use cases with the WFE SuD. This setting page is a frontend for the `NavigationConstants` class (more in paragraph 3.4.5). The generated pages can be found in overview figure 3.10.

3.4.3 Backing beans

All triggering WFE entity action pages are bound to the use case BB and all the X pages are bound to the use case X BB. The backing bean class contains all variables which are entered in the use case with getters and setters. The use case BB contains also methods for delegating methods to the business logic part via the use case BD.

3.4.4 Business delegator

Business delegator is generated as the Business Delegate pattern [20]. Every use case has its own BD which enables call to the use case EJB; the X BD is also generated for the WFE SuD use cases. The Business delegator method calls `ServiceLocator` for a JNDI look up for the EJB class. More about the JNDI look up can be found in [4]. The `ServiceLocator` class is created as the Service Locator pattern [20].

3.4.5 EJB

Beans contain methods which are called from the backing beans on the SuD entity. The method body differs whether the use case SuD entity is WFE or it is not.

Use cases with WFE SuD

The method body belonging to the procedure call trigger action from the use case with the WFE SuD returns the navigation string to pages of the SuD actions. If the procedure call internal actions are starting with a branch action, then condition labels are extracted from the branches which belong to sub-variations or extensions. These labels are added to the `NavigationConstants` class and from this conditional labels is created the *if () ... else if ()* construction where inside the *if* construction is returned a navigation string to that branch page. The last *else* construction belongs to the main scenario and it returns a navigation string to the main scenario branch action page. The similar construction is used when a loop action is the first action inside the procedure call.

Example of a procase procedure call with the WFE SuD entity and the generated EJB method for the trigger action:

$$?a\ \{b; c + d; e + f\}$$

```
public String a(...) {
  if (navigation.b)
    return "nav_to_c";
  else if (navigation.d)
    return "nav_to_e";
  else
    return "nav_to_f";
}
```

The method body belonging to a procedure call internal action from the use case with the WFE SuD contains only a delegation to an action triggering entity manager and the appropriate navigation string is returned.

Use cases with nonWFE SuD

The method body belonging to the procedure call trigger action from the use case with the nonWFE SuD contains the internal procedure call structure. After the Preprocessing step (see 3.2), Procase can recall a common programming language. The branch action can be generated as the *if* condition. The loop action can be generated as the *while* cycle, actions can be generated as methods. The code generation is described more detailed in following paragraphs.

The branch action is generated as *if () ... else ...* construction. The first action inside the branch which refers to the extension or sub-variation is a condition label and it is the condition for triggering this branch, see 2.4. This special fact is not included in the syntax of the Procase language where the condition label is created as the first internal action in the branch. The condition label is generated as a constant variable inside the *if* command clause. The variable declaration is generated apart inside the global static `NavigationConstants` class (as default with value `false`). The `NavigationConstants` class determinate the internal process of all the project entities. Every branch action can be generated as *if () ... else if () ... else ...* construction with only one default (*else*) branch. This default branch is corresponding to the main scenario and it can be at the most one branch in every branch action. Other branches are corresponding to extensions or sub-variations and have a condition label.

The loop is generated as the *while* command. The iteration count is generated as the post decrement construction with the variable made from the use case name. The variable declaration is generated inside the global static `NavigationConstants` class (with default value 3 for 3 iterations). The number of iteration can be defined at runtime via the generated application web front end or it can by changed in the `NavigationConstants` class.

Each action is generated as a delegation to the action triggering entity manager. For the SuD internal actions, methods are called on the use case SuD entity manager and for request send actions, methods are called on the action triggering entity manager. The *%ABORT* action is generated as the return statement `return "abort"`. The conditional event variable declaration is generated as a return statement that returns the conditional event variable label as a string `return "LABEL"`. The unsetting conditional event variable declaration is generated as a return statement which returns the navigation string to the next procedure call trigger action. At the end of the method it returns the navigation string (`"continue"`) which leads to the next PA triggering action page.

The following example shows a procase with a procedure call and a corresponding generated EJB method.

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

```
public String submitItemDescription(String sellerBillingInformation,
   String itemDescription) {
   if (NavigationConstants.priceAssessmentAvailable) {
      getSellerManager().providePriceAssessment();
   }
   else {
      getComputerSystemManager().validateDescription(itemDescription);
      if (NavigationConstants.validationPerformedSystemFails) {
         return "abort";
      }
   }
   return "continue";
}
```

The *INIT* procedure call is generated as `init` method inside the use case EJB and is called as the first command inside the first following procedure call.

The `Local` and the `LocalHome` interfaces are generated for all beans and all EJBs are recorded in the EJB xml files.

3.4.6 Entity Managers

The entity manager refers to the entity itself. The methods belonging to the entity internal actions are generated in the entity manager. The methods belonging to the request send actions called on the entity are also generated in the entity manager. Interfaces for future replacement are generated for all entity managers. The entity manager is called from the EJB tier via a `ManagerLocator` class which is implemented as the Service locator pattern[20].

The calls to the following tiers (e.g. Database Access Objects) should be implemented in the entity manager methods. The implementation of these methods should cover the internal entity logic of the system. The internal entity logic is not handled in the Use cases generally and therefore cannot be generated. For the testing purpose, the implementation of the MGR method is only printing the action label to the server console.

3.4.7 Navigation

The navigation between pages is the important part of the application internal logic. It determine the order of procedure calls and for the WFE SuD, it determine the sequence of actions inside a procedure call. The navigation consist of the set of the navigation rules. The navigation rule has a name of a page to apply on and pairs of navigation string and the name of a page where to go in that case. The navigation string is obtained as a return string from the submitting method in the BB. The navigation rules are located in the *faces-config.xml* file. Overview figure 3.10 shows a navigation from the index page to use case action pages.

The navigation rules are extracted from the project procase. The navigation depends whether the use case has the WFE SuD or not. The nonWFE SuD use cases are processed in the following way.

Each use case and each procedure call is searched for abort actions and for conditional event labels. The navigation rule is created for the trigger action page. The navigation can lead to the abort page in case of the abort action is located inside the procedure call or the navigation can lead to the end of the use case (entity page) or it can lead to the following procedure call (following procedure trigger action page). The following procedure calls which are located inside a branch beside the *NULL* action are not allowed because they may not be proceeded. In case that the conditional event label

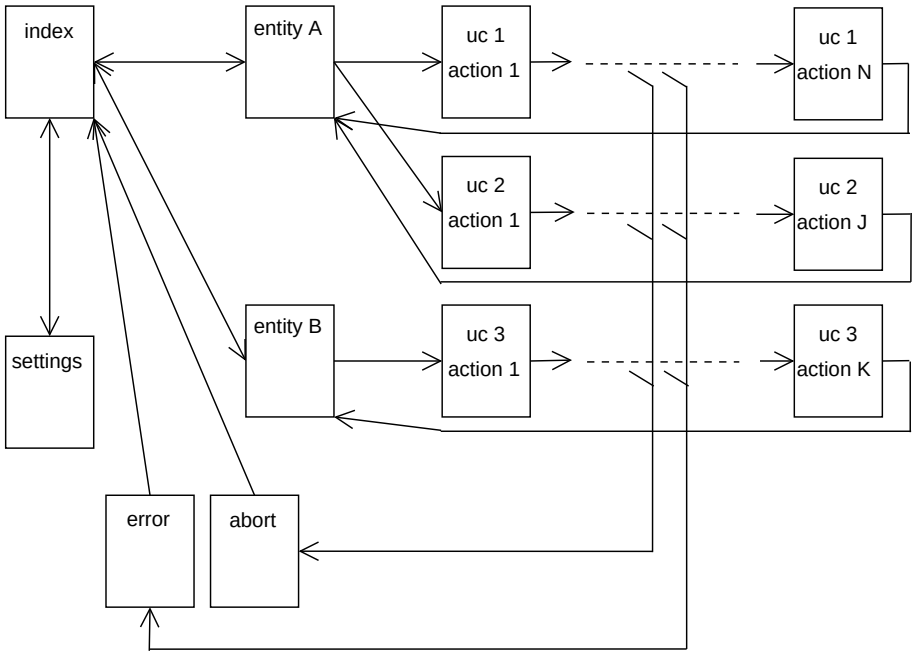


Figure 3.10: The navigation between the generated pages.

is located in a procedure call then the navigation rule leads to a following procedure call trigger action page where the procedure call is located inside the marked branch with with the same conditional event variable and beside the *NULL* action.

The following example shows the procase and its navigation (with non-WFE SuD).

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

$$\begin{array}{lcl} \text{a} & \rightarrow & \text{End D} \\ \text{d} & \rightarrow & \text{P H} \\ \text{p} & \rightarrow & \text{End} \\ \text{h} & \rightarrow & \text{End} \end{array}$$

The navigation rules are more complicated when the use case has the WFE SuD. In this case, there are three types of the navigation rules. The first type is the navigation from a procedure trigger action page to actions pages inside the procedure call. The second type is the navigation from a action page to actions pages inside the procedure call. The third type is the navigation from a action page inside the procedure call to the next procedure call trigger action page.

Each type is described in the following paragraphs:

1. The navigation from the procedure call trigger action page leads to the first action page inside the procedure. If the procedure call contains a branch action or a loop action as the first action then all possible first action pages are included in the navigation rule. If a branch action is the first action inside the procedure call then the first actions are all the first actions in each branch. If a loop action is the first action inside the procedure call then the first actions are the first action in the internal loop actions and the following action after the loop action.

2. The navigation between action pages inside the procedure is similar to the previous type. The navigation leads to the next action page. If the next action is a branch action or a loop action then all the possible following action pages are included in the navigation rule.

3. The navigation from the action page inside the procedure call to the next procedure call trigger action page is a simple navigation role.

The navigation rules of the type 1 and 3 are for further application development redundant and they have just a testing purpose which enable to navigate through the use case as a one user (changing roles = entities). Anyway the client-server architecture does not allow a meaningful direct communication between two WFE entities. The following example shows a procase and its navigation (with WFE SuD).

$$?a\ (b; c + d; \$E);\ (E\ :?e\{f; (\%ABORT + g)\} + NULL);\ ?h\ \{i\}$$

a	$\rightarrow$	b d
b	$\rightarrow$	c
c	$\rightarrow$	h
d	$\rightarrow$	e
e	$\rightarrow$	f
f	$\rightarrow$	Abort g
g	$\rightarrow$	h
h	$\rightarrow$	i
i	$\rightarrow$	End

3.4.8 Supporting objects

Core objects

The core objects refer to the conceptual classes in the project domain model (see 2.3) and are used as arguments in generated methods. This work is not focusing on this part of the generation process and therefore the simplest way of generation is chosen, see future work 5.2.

The conceptual classes are generated as java classes and the conceptual classes attributes are generated as its attributes with the assigned type; default type is generated String. The conceptual classes relations are not processed and its generation is discussed in the future work of the thesis see 5.2.

Session objects

The session object (object stored and assigned to one session) is generated for the use case which uses one variable in more then one procedure call. The session object stores variables which has to be permanent during procedure calls. The variables were marked as session arguments during the Determining arguments step of the proposed process, see 3.3.2. The session object is stored in a faces session map and obtained in the use case BB when needed.

3.4.9 Other issues

Method names conflicts

The generated methods (except for the MGR tier) have a names created from the SuD entity name, the use case name and the procedure call name; connected together with the underline (_). For example, the action *#validateItemDescription* located inside the *Seller to Clerk* use case within the *submitItemDescription* procedure call and with the *Clerk* as the SuD entity has the method name `Clerk_SellerToClerk_submitItemDescription_validateItemDescription`. Methods generated in the entity MGRs has name created only from the actions token; in this case *validateItemDescription*.

The action labels should be in Procase unique as presumed in subsection 3.4.10. The actions with same labels in the use case are generated as one method in the BD and in the EJB but it is necessary to have unique method name in the BB. The methods names in the BB need to be different

because the returned navigation string may be different for each action according to the following actions. In this case, the suffix number is appended to the conflict method name. The duplicated actions can be brought to the procase during the Preprocessing, see 3.2.

The following example shows the procase before and after the Preprocessing where the first procedure call a has two same actions f and altogether there are three actions f:

$$?a; b; (c; ?d; e + u; v; w + x; y; z); f$$

$$?a\ \{b; (c; \$D + u; v; w; f + x; y; z; f)\};\ (D\ : ?d\ \{e; f\} + NULL)$$

Linked use cases

The *Linked use cases* are use cases when a request send action token match a request receive action in another use case. The proposed application construction allows linked use cases, therefore the business logic was moved from the BB to the EJB tier. In case that a request send action matches to a procedure call trigger action then a delegation to the linked use case procedure call can be triggered. The action method and the procedure call trigger action must match with a name and a signature. In case that they match with the name and does not match with the signature then a comment is generated where the possible linked procedure call trigger action is mentioned. The procedure call trigger action as the use case EJB is called in the MGR action method via the BD. The described construction is applied only when the linked use case has nonWFE SuD because the linked use case with the WFE SuD has actions represented as web pages and the application cannot be redirected to the web pages.

The linked use cases construction is based on matching action tokens therefore it depends on carefully created procases. The proposed process links only to procedure calls which has the nonWFE SuD. Anyway, the generated manager methods should be reviewed.

The similar construction is *Included use cases*, where the Procase language is enhanced with special action which represents included use case. The Procasor program (see 2.5) does not support this special construction thus it is not supported in the proposed generator.

Exceptions

The abort action (*%ABORT*) is not a typical trace of the program, but it is described in this early phase of designing the system (while writing the use cases) and therefore it is generated as a normal trace of the program. The abort action is not generated as an exception, but rather as a different return value. Exceptions could be added to the generated project for the unexpected and uncommon traces of the program which are not expressed in the use cases.

Different approach is to add exceptions to Procase language syntax (see [14]). This Procase language extension with the hard semantic layout is more difficult to process by computer and the Procasor program, see 2.5, does not generate Procase with this extension and therefore this exception construction is not supported in this work.

Packaging

The application packaging to the deploy-able package (see Appendix B.3) is maintained by the Ant build [19]. The packaging differs according to the chosen application server. The proposed generator produces XML descriptors for deploying to the JBoss application server [1]. Create or change the descriptors for different application server should not be difficult.

Use case expressions

The Use case expression (see 2.4.2) indicates a behavior of an entity and determines possible traces of use cases triggered on the entity. If the use case represents a service which is offered to the involved entity, then the consideration of these services from the client-server view is needed.

For a client-server application, all the services should be offered parallel, therefore the parallel run operator (||) has no important meaning and it should be implemented within a task manager in the intended application. The services should be offered repeatedly, therefore the iteration operator (*) has no important meaning either. The alternative operator (+) refers to the individual suspending and should be implemented with the locking over database tables or rows or inside a login module.

The only use case expression operator which should be handled in the proposed process is the sequence operator (;). This issue is discussed in the

future work 5.2. The use case expressions are not handled in the proposed generator program.

3.4.10 Restrictions

A project written in the Procasor program (see 2.5) allows several *Entity models* for every entity. The entity model is set of use cases belonging to the entity and it allows describing tasks of the entity in different points of view. However, the proposed generating process does not solve possible duplicated use cases in different models. Therefore, the only one model is allowed; more about choosing the entity model can be found in Appendix B.2.

Also, a project written in the Procasor program allows to describe the entity hierarchy where the child entities have their detail use cases. However, the proposed generating process uses only use cases belonging to the most detailed entities (leafs in the entity hierarchy tree) and does not solve possible duplicated use cases. The generator program also does not solve possible conflicts between use cases which are describing same activity but with different SuD entities (from different points of view).

The proposed generating process also assumes that the action tokens are unique within the project if it is unique action and on the other hand action tokens match if they express the same action. The Procasor program [18] has the token manager to help solving this issue.

Chapter 4

Case study

4.1 Marketplace project overview

The Marketplace project [12] is used as a case study. Originally the project contained only use cases; the domain model has been created as a part of the thesis.

The *Marketplace* is an application for online buying and selling like the *ebay*. There are *Seller*s who want to sell things and therefore they enter an offer to the system. Also there are *Buyer*s who are searching for an interesting offer. The Seller and the Buyer communicate directly with the *Computer system*. In two cases they communicate through the *Clerk* who is passing information to the Computer system. There is also the *Supervisor* who maintains the Computer system. The *Credit verification agency* enables sellers and buyers to operate in the system and the *Trade commission* permits each offer to be sold. The entities and the Marketplace entity hierarchy are captured in figure 4.1 where entities represented with a figure are the WFE entities.

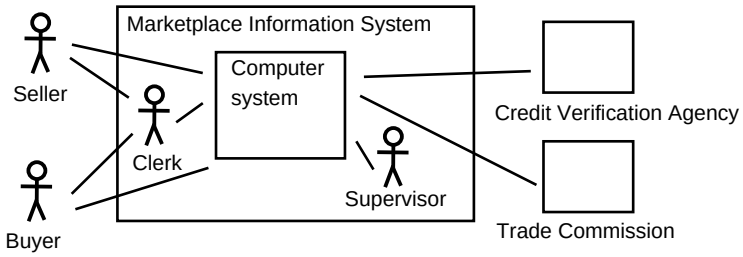


Figure 4.1: The entity hierarchy in the Marketplace project.

4.1.1 Marketplace use cases

The Marketplace use cases are described in in the following paragraphs (in brackets is reference to Appendix where the use case and the procase are included). The complete listing is available on enclosed CD, see Appendix C, and for brief overview the use case diagrams are shown in the following figures:

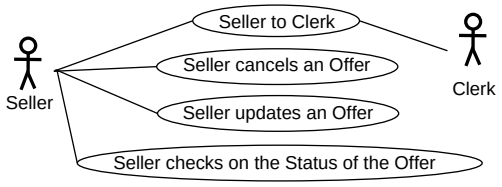


Figure 4.2: Seller use case diagram

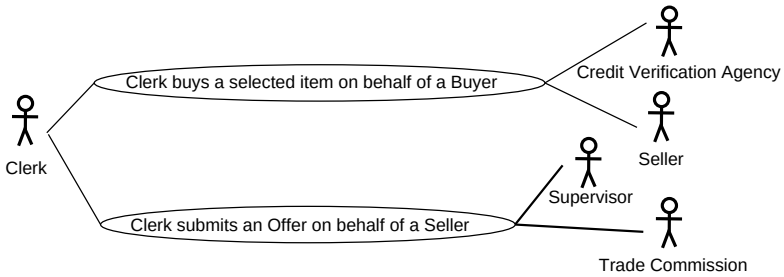


Figure 4.3: Clerk use case diagram

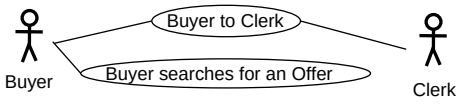


Figure 4.4: Buyer use case diagram

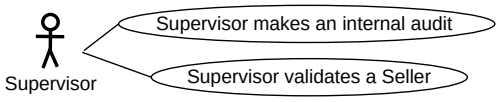


Figure 4.5: Supervisor use case diagram

The Seller can enter (C.8) or cancel (C.4) or update (C.6) an offer or check (C.5) the status of the offer. Figure 4.2 shows the Seller use case diagram. The Buyer can search for an interesting offer (C.2) or can buy the selected offer (C.9). The Buyer use case diagram is captured in the figure 4.4. The Clerk guides a Seller while is entering an offer (C.1) and also a Buyer while is accepting an offer (C.3). Figure 4.3 represents the Clerk use case diagram. The Supervisor can validate a Seller (C.10) or make an internal audit (C.7). The Supervisor use case diagram is captured in figure 4.5. Table 4.1 shows the use cases with involved entities.

	use case name	SuD	PA	other act.
0	Buyer to Clerk (C.8)	CL	B	CS
1	Seller to Clerk (C.9)	CL	SL	CS
2	Buyer searches for an Offer (C.2)	CS	B	
3	Clerk buys a selected item on behalf of a Buyer (C.3)	CS	CL	SL, CVA
4	Clerk submits an Offer on behalf of a Seller (C.1)	CS	CL	TC, SV
5	Seller cancels an Offer (C.4)	CS	SL	
6	Seller checks on the status of the Offer (C.5)	CS	SL	
7	Seller updates an Offer (C.6)	CS	SL	
8	Supervisor makes an internal audit (C.7)	CS	SV	
9	Supervisor validates a Seller (C.10)	SV	CS	

Table 4.1: SL = Seller, B = Buyer, CL = Clerk, CS = Computer System, SV = Supervisor, TC = Trade Commission, CVA = Credit Verification Agency

The Marketplace project contains 19 use cases, but only use cases which are processed by the generator are presented in Table 4.1. The use cases which have *the Marketplace Information System* as the SuD entity are not processed by the generator, because the Marketplace Information System entity has

child entities and there can be duplicated use cases, see Section 3.4.10. All project use cases can be found in Appendix A. The Marketplace procases before and after Preprocessing can be found in Appendix C.

4.1.2 Marketplace domain model

The domain model contains five conceptual classes: *Seller*, *Buyer*, *Offer*, *Item* and *Payment*. The domain model contains following conceptual classes relations. A Seller have *submitted* some Offers and an Offer *consists of* Items. A Buyer have *paid* some Payments where the Payment *covers* an Offer. All the conceptual class attributes are captured in the Marketplace domain model, see figure 4.6.

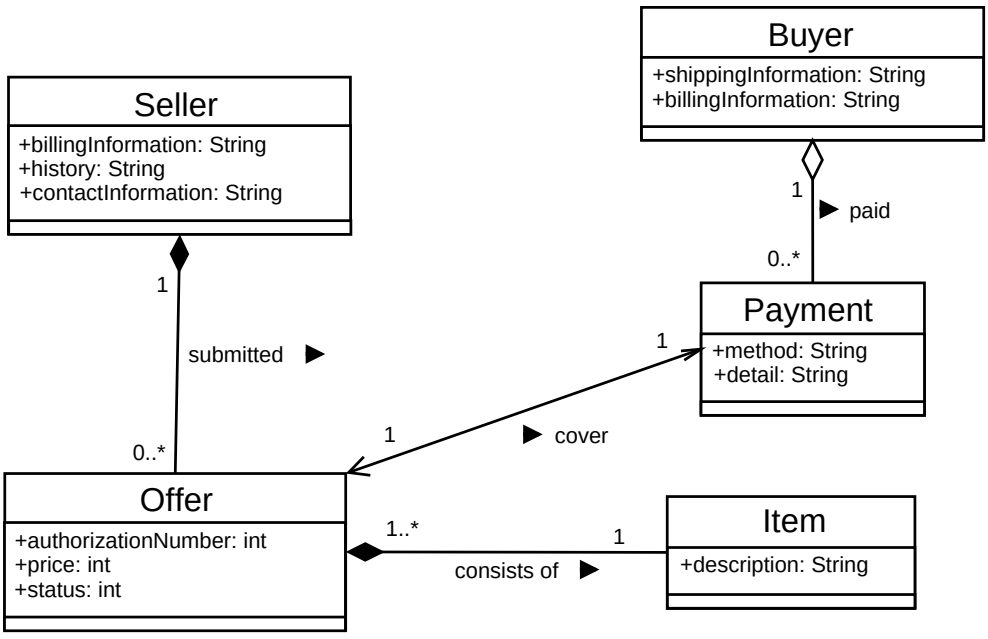


Figure 4.6: Marketplace domain model

The domain model was created by this work from the noun phrases found in the Marketplace use cases; this technique is described in [11].

4.2 Generated elements

The use cases number 0, 1 and 9 (numbers according to Table 4.1) have the WFE SuD therefore the internal logic structure is generated as frontend

(X pages, X BB, X BD, X EJB, see 3.4.1). The rest of the use cases has the nonWFE SuD therefore the internal logic structure is generated as methods inside the use case EJB.

Use cases with the WFE triggering entity can be triggered via the web frontend, only the Computer System is the nonWFE entity, hence the use case number 9 cannot be triggered via the web frontend. Entities and use cases which can be triggered by each entity are captured in Table 4.2.

entity	triggered use cases
Seller	1, 5, 6, 7
Buyer	0, 2
Clerk	3, 4
Supervisor	8
Computer System	9

Table 4.2: Triggering entities

4.2.1 Example of generated elements

For ilustration, generated elements of a one Marketplace use case are described in more detail in following paragraphs. The use case *Clerk submits an offer on behalf of a Seller* (full use case can be found in C.1) starts with following use case steps:

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 following procase is created after the Preprocessing:

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

The use case from which is this use case fragment has the nonWFE SuD entity *Computer System*, the WFE PA entity *Clerk* and three supporting actors which does not take part in this fragment. The generator program creates for this procase fragment one JSF page, one method inside use case BB, one method inside use case BD and one method inside use case EJB. Methods in the MGR are generated apart. The internal logic is located in the created use case EJB because it is nonWFE SuD use case. The created methods and code fragments are listed in following extracts and described in following paragraphs.

Part of the JSF page `ComputerSystem_ClerkSubmitsAnOfferOnBehalfOfASeller_submitItemDescription.jsp`:

```
<h:form>
  <h:outputText value="itemDescription0 : "  />
  <h:inputText id="itemDescriptionZero"
    value="#{ComputerSystem_ClerkSubmitsAnOfferOnBehalfOfASeller_
            ClerkBBean.itemDescriptionZero}" />
  </br>
  <h:outputText value="sellerBillingInformationZero : "  />
  <h:inputText id="sellerBillingInformationZero"
    value="#{ComputerSystem_ClerkSubmitsAnOfferOnBehalfOfASeller_
            ClerkBBean.sellerBillingInformationZero}" />
  </br>
  <h:commandButton value="submitForm"
    action="#{ComputerSystem_ClerkSubmitsAnOfferOnBehalfOfASeller_
            ClerkBBean.submitItemDescription}" />
</h:form>
```

The JSF page has a simple form, an input field for action argument and a submit button. The triggering action `submitItemDescription` has one argument `itemDescription0`. The generated page has for this argument an input field which is bound to a use case backing bean variable `itemDescriptionZero`. The variable name is changed because JSF [22] name framework rules.

Part of the backing bean class `ComputerSystem_ClerkSubmitsAnOfferOnBehalfOfASeller_Clerk.java`:

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

The use case BB contains method which is triggered when a user submits the form on the page. Before the method is triggered values in input fields inside submitted form are set to bounded variables. The variable is used as argument for the BB method and is called the EJB method via the use case BD.

Part of the business delegator class `ComputerSystem_ClerkSubmitsAnOfferOnBehalfOfASeller.java`:

```
public String submitItemDescription(String sellerBillingInformation0,
  String itemDescription0, ComputerSystem_
    ClerkSubmitsAnOfferOnBehalfOfASellerSO sessionObject) {
  try {
    return getBean().submitItemDescription(sellerBillingInformation0,
      itemDescription0, sessionObject);
  } catch (BeanException e) {
    throw new DelegateException(this.getClass().getName() +
      ".submitItemDescription()", e);
  }
}
```

The BD method only delegates calls to the use case EJB. The `getBean` method contains code for obtaining a use case bean `LocalHome` interface and creating the bean and returns a use case bean stub.

Part of EJB `ComputerSystem_ClerkSubmitsAnOfferOnBehalfOfASeller.java`:

```
public String submitItemDescription(String sellerBillingInformation0,
  String itemDescription0, ComputerSystem_
    ClerkSubmitsAnOfferOnBehalfOfASellerSO sessionObject) {
  if (Constants.computerSystem_
    ClerkSubmitsAnOfferOnBehalfOfASeller_
      priceAssessmentAvailable) {
    getSellerManager().providePriceAssessment();
  }
  else {
    getComputerSystemManager()
      .validateDescription(itemDescription0);
    if (Constants.computerSystem_
      ClerkSubmitsAnOfferOnBehalfOfASeller_
        validationPerformedSystemFails) {
      return NavigationConstants.ABORT;
    }
  }
  return NavigationConstants.CONTINUE;
}
```

The EJB method contains internal logic; 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 which delegates call to the particular MGR.

4.3 Evaluation of the generated project

4.3.1 Determined method arguments

The extracted methods arguments are not always intended and must be reviewed. However the success ratio of the successfully extracted methods arguments in Marketplace project is pleasant. The case study project has 92 actions where in 12 cases there are arguments obviously missing and the misleading arguments were extracted in 4 cases. In other cases, the generated method arguments possibly will not be used in the final application, but they are mentioned in the use case step sentence. This proves that the Determining argument step of the proposed process, see 3.3, is well designed though may seem to be simple. The possible improvement of the Determining argument step is discussed in the future work 5.2.

4.3.2 Missing elements

The generated application frontend can be investigated for missing elements of the project use cases more interactively than reading the use cases. However, no major missing elements were found in the Marketplace project. The found missing elements are mentioned in following paragraphs.

In the Marketplace project, one use case is missing. The offers can be entered, updated, sold or canceled by the seller. But there should be another use case where supervisor will discard not sell-able offers after a certain time.

Use case number 9 does not reflect case that the supervisor can deny the Seller to operate in the application. This missing branch cannot be tested with the generated frontend because it has the nonWFE triggering actor which is not supported by the prototype generator.

4.3.3 Linked use cases

Linked use cases (see 3.4.9) in the Marketplace project are intended in 6 places and Table 4.3 shows the linked use cases with its actions tokens.

	uc from	"!" action token	uc to	"?" action token
1	0	validateSeller	9	validateSeller
2	7	submitItemDescription	0	submitItemDescription
3	7	enterPriceBilling-ContactInformation	0	enterPriceContact-BillingInformation
4	8	submitSellectOffer	2	acceptSelectOffer
5	8	enterBillingShipping-InformationPaymentMethod-PaymentDetail	2	enterBillingInformation-PaymentMethod
6	9	permitSeller	0	permitSeller

Table 4.3: Linked use cases (uc = use case; number in column refers to use case number in Table 4.1)

In cases 1 and 6, there are the linked use case with the WFE SuD entity and therefore the proposed generator skips these cases. In cases 3, 4 and 5 the action tokens does not match and they should be corrected. In case 2,

the proposed generator was not able to link the use cases because the actions does not match with signature; the procedure call trigger action (method) has another argument.

The proposed generator have failed to link the use cases mainly due to the imprecision of the input procases.

4.3.4 Future usage of generated application

The Marketplace application is generated with no flaws which disable to test it. The use cases which has WFE SuD can be tested for missing elements in the system requirements. The generated project is not complete application and at least following parts must be further developed.

The generated project does not have the backend tiers. The Data persistence tier and the Data access tier need to be developed to obtain more advanced implementation model. The proposed generator is not focusing in the generation of this tiers, see related work 5.1 for tools taking care of this issue.

The generated frontend has only the testing purpose and has to be redesigned, more in 5.2. Login module implementation is necessary to separate and secure the entities - roles.

Chapter 5

Conclusion and Future work

5.1 Related work

Today there are many generators that can generate source files, database tables and/or xml descriptors from the Class diagram or from the Domain model. The major difference between these generators and the proposed process is that the proposed process handles the use cases – the communication of the entities. Other generators produces a part of program which allows only to access/change/store objects according to the Class diagram structure but the sequences of the actions which are captured in the Use cases are not handled. The proposed generator can be directly compared with other generators only from the aspect of generating the objects from the Domain model (see 3.4.8), but this work is not focusing on that part of generation.

There are many other generators which generate source files from the Class diagrams (almost every case tool can export the Class diagrams) but in the following paragraphs are mentioned generators which were investigated by this work and which are in some way interesting.

The AndroMDA [3] (pronounced "Andromeda") is the generator framework which transforms the UML models into the implemented components. Today the transformations are written for many technologies, like Spring, EJB, .NET, Hibernate and Struts or it is possible to write the new transformation (so-called cartridge). The AndroMDA has modular design that enables to change each major generator components and produce intended output. In basic the AndroMDA works with the UML Class diagrams and according to the class stereotype it generates the corresponding output files.

The Enterprise Core Objects [5] (introduced in [9]) is a framework that uses a UML model for generating the runtime data persistence framework. This framework is a solution for .NET and the ECO is integrated

in C#Builder and also in Delphi 8. This project demonstrate the seamless integration of the Class diagram analysis into the programming process.

The Jamda [10] is a framework for building the application generators which create a java implementation of the Domain model. It generates java source file according to the model class stereotype. The framework provide a flexible structure to meet the custom demands.

5.2 Future work

An interesting possibility of future extension is the integration of the Procasor program (see 2.5) and the proposed generator into the AndroMDA [3] generator framework. The possible massive usage can brought the integration of the Procasor program and the proposed generator into the AndroMDA eclipse plugin [8].

An unresolved issue is an instance handling inside the Use case. The proposed process handles the recognized arguments in the simplest way and therefore there is lot of to improve.

The relations between the Conceptual classes inside the Domain model are not handled in the generation process. The relations refers to collections, lists or arrays in a programming language, therefore the participation of these objects in the phase of determining action arguments is not simple as for the conceptual classes and its attributes. More investigation about this issue should proceed.

Another interesting issue is to work with verb types while recognizing the method arguments. The verbs used in the use case sentences will be categorized and the arguments can be assigned as input or output according to the matched verb category.

The use case expressions are another unresolved issue, mainly the sequence operator. The sequence of the use cases should be reflected by the frontend and allow the WFE entity to trigger the use cases only in that sequence. This issue may be connected with the future work issue about the instance handling inside the Use case.

The proposed generator may be changed to use the custom template web pages. Then the generated web pages will have more intended outfit and could be used for tuning the frontend model screens.

5.3 Conclusion

The process proposed by this thesis allows to generate a test ready application along with a project file structure of the implementation model. A proof of the concept, a tool taking analyzed use cases and domain model as an input and producing the implementation model project has been successfully developed.

The generated application can be used for testing use cases and can give a feedback to the project's analysts to see whether there are some elements missing in the system requirements. The testing web frontend gives a good orientation in what is happening in the application and what continue according to the use case.

The usable project skeleton structure is generated based on the present practice for the projects developed with the J2EE technologies. The generated files are structured in logical packages and are easy to use. The generated application structure is prepared for future development.

Additionally, this work analyzed and implemented the algorithm of completing the Procase language with the procedure calls brackets, see 3.2. The Procase completed with the procedure calls brackets is necessary for the proposed process. The algorithm for procases with the certain complex structure employs the Procase enhancement (the Conditional events and the Aborts actions) which is elaborated and formulated in this thesis. The proposed algorithm and enhancement might be interesting for other researches where the Procase language is processed.

Bibliography

[1] JBoss a division of Red Hat. Jboss application server. `http://labs.jboss.com/`.

[2] Bente Anda and Dag I. K. Sjober. Investigating the role of use cases in the consturction of class diagrams. *Empirical Software Engineering*, page 24, 2005.

[3] AndroMDA. Andromda: Extensible generator framework. `http://galaxy.andromda.org/`.

[4] Rosanna Lee at Sun Microsystems. Building directory-enabled java applications. `http://java.sun.com/products/jndi/tutorial/`.

[5] Borland. Enterprise core objects. `http://www.borland.com`.

[6] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Pub Co, first edition, 2000.

[7] Jaroslav Drazan. Natural language processing of textual use cases. Master's thesis, Charles University, Prague, Czech Republic, 2006.

[8] Eclipse. Eclipse - an open development platform. `http://www.eclipse.org/`.

[9] Malcolm Groves. Introduction to enterprise core objects (eco). *Borland Conference*, 2004.

[10] Jamda.net. Jamda: open-source framework for building application generators. `http://jamda.sourceforge.net/`.

[11] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall PTR, second edition, 2001.

[12] Vladimir Mencl. *Use Cases: Behavior Assembly, Behavior Composition and Reasoning*. PhD thesis, Charles University, Prague, Czech Republic, 2004.

[13] OMG. Unified modeling language: Superstructure, verision 2.0, final adopted specification.
`http://www.omg.org/uml/`.

[14] Frantisek Plasil and Viliam Holub. Exceptions in component interaction protocols - a necessity. In Ralf Reussner, Judith A. Stafford, and Clemens A. Szyperski, editors, *Dagstuhl Seminar 04511 - Architecting Systems with Trustworthy Components*, volume 3938 of *Lecture Notes in Computer Science*, pages 227–244. Springer-Verlag, Jan 2006.

[15] Frantisek Plasil and Vladimir Mencl. Getting "whole picture" behavior in a use case model. *Transactions of the SDPS: Journal of Integrated Design and Process Science, vol. 7, no. 4*, 2003.

[16] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *Transactions on Software Engineering, IEEE*, 28(11), 2002.

[17] Frantisek Plasil, Stanislav Visnovsky, and Miloslav Besta. Behavior protocols, 2001.

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

[19] The Apache software fundation. Apache ant, a java-based build tool.
`http://ant.apache.org/`.

[20] Inc. Sun Microsystems. Core j2ee patterns: Best practices and design strategies.
`http://java.sun.com/blueprints/corej2eepatterns/`.

[21] Inc. Sun Microsystems. Java platform, enterprise edition (java ee).
`http://java.sun.com/javaee/`.

[22] Inc. Sun Microsystems. Javaserver faces technology, java platform, enterprise edition (java ee).
`http://java.sun.com/javaee/javaserverfaces/`.

Index

Appendix A

Content of CD

The enclosed CD contains:

directory name	description of content
src	source files of the proposed generator
externals	external jars used by the generator
templates	template files used during the generation
Marketplace	generated example project
marketplace.txt	text file with Marketplace use cases and procases
inputFiles	input files – Marketplace.pro (analyzed example project), DomainModel.xsd (schema for domain model xml file), entityTypes.txt (entity type config file), MarketplaceDomainModel.xml (example domain model).
gen.sh	script for executing the generator
gen.jar	generator jar file

Appendix B

Proof of concept prototype generator

Used technologies for the implementation model are Java 1.4.2, JSF 1.2 [22], JBoss 1.4.0.3 [1] and Ant 1.6.5 [19]. The generator is implemented with following technologies Java 1.4.2, JAXP 1.3. The greater versions of the used technologies might be also sufficient however they have not been tested.

B.1 Input files

The input files for the implementation model generator are following files:

project xml file *	The analyzed project saved by the Procasor program, see 2.5. Contains the analyzed use cases and procases.
domain model xml file *	The domain model with semantic described by the DomainModel.xsd file (see A).
entity type config file	The config file to determine which entity is WFE or not (see 2.7).
model chooser config file	The config file to determine which project use case model will be loaded, see 3.4.10.

The input files marked with a star (*) are mandatory. The config files must have semantic *name = value*.

B.2 Running generator program

The proof of concept prototype implementation of the implementation model generator can be executed by the shell script `gen.sh`. However the bat file for executing the generator can be created similar to the shell script.

The generator program synopsis:

```
./gen.sh [-v] -p FILE -d FILE [-e FILE] [-m FILE]
    ./gen.sh [-v] -t [-e FILE] [-m FILE]
```

option syntax	option description
-v	The generator is verbose.
-p FILE	The generator loads the project from the file FILE.
-d FILE	The generator loads the domain model from the file FILE.
-e FILE	The generator loads the entity type config file from the file FILE.
-m FILE	The generator loads the model config file from the file FILE.
-o	The generator process only the Preprocessing step (see 3.2).
-t	The generator loads the Marketplace example project file and the case study domain model file.

B.3 Testing generated project

To run the generated application it is required to run the JBoss [1] application server then it might be appropriate to edit the build.properties file and run the Ant [19] build script with a default target. Once the application is deployed then the index page can be visited on the address `http://localhost:8080/`*project_name*`/index.faces`. Detail information about the JBoss application server and deployment can be found in [1].

B.4 Generator program description

Source code files of the generator program are available on enclosed CD (see Appendix A) and are written with javadoc comments and are structured in several packages. The description of the program process is described in following paragraphs.

The generator source file are in the *src* directory with the generator main class `Gen` which runs steps of the generator process; described in 3.1. The generator program starts with parsing the input files, see B.1. The project xml and the domain model files are parsed with JAXB technology. The entity type and the model config files are parsed afterward. The input parsing source code files are located in *inputparser* package.

The next phase is used for generation of names which will be used for generated elements, see 3.4.9. The domain model is checked for attribute name uniqueness and names used in the Domain model are transformed to Java name convention. The analyzed use case project (created by the Procasor program 2.5) is checked for entity name uniqueness and the action names are also transformed to Java name convention.

Then the Preprocessing is applied on the input where the inputted procases are enhanced with procedure call brackets, for more information see 3.2. The Preprocessing source code files are located in *preprocessing* package. After this phase the generator can end with printing out the enhanced procases, see B.2 or continue with application generation. The methods arguments are determined afterward, see 3.3, which are used in generated application.

The generator continues with application generation, see 3.4. Source code files which are employed for generation are located in *generator* package; with `MainGenerator` main class. First, the directory tree is created and template files are copied from the template directory. The template files are modified to fit in the project with a package name and an absolute path. Then the application source files are generated, for more information about generated files see corresponding paragraph in 3.4. The generation of the source code files is managed by `SudNonWfeGenerator` class for use cases with the nonWFE SuD entity and by `SudWfeGenerator` class for use cases with the WFE SuD entity. Both classes are located in *generator* package. Lastly, the ANT build files are created. At this point, the generator program ends and the generated application project can be builded and deployed.

Appendix C

Marketplace use cases and procases

C.1 Clerk submits an offer on behalf of a Seller

Use case

UseCase: Clerk submits an offer on behalf of a Seller

Scope: Marketplace
SuD: Computer System
Primary actor: Clerk
Supporting actor: Trade Commission
Supporting actor: Supervisor
Supporting actor: Seller

Main success scenario specification:
1 Clerk submits information describing an item.
2 System validates the description.
3 Clerk adjusts/enters price and enters seller's contact and billing information.
4 System validates the seller's contact information.
5 System asks the Supervisor to validate the seller.
6 Supervisor permits the seller to operate on the marketplace.
7 System validates the whole offer with the Trade Commission.
8 System lists the offer in published offers.
9 System responds with an uniquely identified authorization number.

Extensions:
2a Validation performed by the system fails.
2a1 Use case aborted.
7a Trade commission rejects the offer.
7a1 Use case aborted

Sub-variations:
2b Price assessment available.

2b1 System provides the seller with a price assessment.

Procase

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

Procase after Preprocessing

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

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

C.2 Buyer searches for an offer

Use case

UseCase: Buyer searches for an offer

Scope: Marketplace
SuD: Computer System
Primary actor: Buyer

Main success scenario specification:
1 Buyer enters basic search criterion.
2 System responds with the list of matches.
3 Buyer requests the complete listing of a selected offer.
4 System responds with the requested information.

Extensions:
2a No matches found.
2a1 Use case aborted.

Sub-variations:
2b The amount of matches is too high.
2b1 Buyer narrows the search results with additional criteria.

Procase

```
?B.enterSearch;
#respondList;
#noMatchFind;
%ABORT
+
```

```
(
?B.enterSearch;
#amountMatchTooHigh;
?B.narrowSearchResult
+
?B.enterSearch;
#respondList
);
?B.select;
#respondInformation
```

Procase after Preprocessing

```
?B.enterSearch {
    (
        #amountMatchTooHigh;
        $NSR
    +
        #respondList;
        (
            #noMatchFind;
            %ABORT
        +
            NULL
        )
    )
};
(NSR:
    ?B.narrowSearchResult {
    }
+
    NULL
);
?B.select {
    #respondInformation
}
```

C.3 Clerk buys a selected item on behalf of a Buyer

Use case

UseCase: Clerk buys a selected item on behalf of a Buyer

Scope: Marketplace
SuD: Computer System

Primary actor: Clerk
Supporting actor: Seller
Supporting actor: Credit Verification Agency
Supporting actor: Buyer

Main success scenario specification:
1 Clerk is contacted by a buyer who has decided to accept a selected offer.
2 System validates the offer.
3 Clerk enters billing information, select a payment method and provides the necessary detail.
4 System validates this information with a Credit Verification Agency.
5 System performs the trade.
6 System informs the seller that the offer has been accepted and provides the shipping information.
7 System transfers the payment to the sellers account.
8 System responds to the buyer with an uniquely identified authorization number.

Extensions:
2a Offer is not valid.
2a1 Use case abort.

Procase

```
?CL.acceptSelectOffer;
#validateOffer;
(
    ?CL.enterBillingInformationPaymentMethod;
    !CVA.validateInformation;
    #performTrade;
    !Sl.informAcceptOffer;
    #transferPayment;
    !B.respondUniquelyIdentifyAuthorizationNumber
+
    #offerNotValid;
    %ABORT
)
```

Procase after Preprocessing

```
?CL.acceptSelectOffer {
    #validateOffer;
    (
        #offerNotValid;
        %ABORT
    +
```

```
            $EBIPM
        )
};
(EBIPM:
        ?CL.enterBillingInformationPaymentMethod {
            !CVA.validateInformation;
            #performTrade;
            !Sl.informAcceptOffer;
            #transferPayment;
            !B.respondUniquelyIdentifyAuthorizationNumber
        }
+
        NULL
)
```

C.4 Seller cancels an offer

Use case

UseCase: Seller cancels an offer

Scope: Marketplace
SuD: Computer System
Primary actor: Seller

Main success scenario specification:
1 Seller locates a previously submitted offer.
2 Seller requests the system to cancel the offer.
3 System responds with a request for the seller to prove identity.
4 Seller responds with the authorization number returned when the offer was submitted.
5 System validates the request and seller's identity.
6 System removes the offer.

Extensions:
4a Seller cannot provide the authorization number.
4a1 Use case is aborted.
5a Authorization number is not valid.
5a1 Use case is aborted.

Procase

```
?Sl.locateSubmitOffer;
?Sl.cancelOffer;
!Sl.requestProveIdentity;
?Sl.respondAuthorizationNumber;
```

```
(
    #sellerCannotProvideAuthorizationNumber;
    %ABORT
+
    #validateIdentity;
    (
        #removeOffer
    +
        #authorizationNumberNotValid;
        %ABORT
    )
)
```

Procase after Preprocessing

```
?Sl.locateSubmitOffer {
};
?Sl.cancelOffer {
    !Sl.requestProveIdentity
};
?Sl.respondAuthorizationNumber {
    (
        #sellerCannotProvideAuthorizationNumber;
        %ABORT
    +
        #validateIdentity;
        (
            #removeOffer
        +
            #authorizationNumberNotValid;
            %ABORT
        )
    )
}
```

C.5 Seller checks on the status of the offer

Use case

UseCase: Seller checks on the status of the offer

Scope: Marketplace
SuD: Computer System
Primary actor: Seller

Main success scenario specification:

1 Seller locates a previously submitted offer.
2 Seller requests the system to provide status of the offer.
3 System responds with a request for the seller to prove identity.
4 Seller responds with the authorization number returned when the offer was submitted.
5 System validates the request and seller's identity.
6 System returns the status of the offer.

Extensions:
4a Seller cannot provide the authorization number.
4a1 Use case is aborted.
5a Authorization number is not valid.
5a1 Use case aborted.

Procase

```
?Sl.locateSubmitOffer;
?Sl.requestProvideOfferStatus;
!Sl.requestProveIdentity;
?Sl.respondAuthorizationNumber;
(
    #sellerCannotProvideAuthorizationNumber;
    %ABORT
+
    #validateIdentity;
    (
        !Sl.returnStatus
    +
        #authorizationNumberNotValid;
        %ABORT
    )
)
```

Procase after Preprocessing

```
?Sl.locateSubmitOffer {
};
?Sl.requestProvideOfferStatus {
    !Sl.requestProveIdentity
};
?Sl.respondAuthorizationNumber {
    (
        #sellerCannotProvideAuthorizationNumber;
        %ABORT
    +
        #validateIdentity;
        (
```

```
                !Sl.returnStatus
            +
                #authorizationNumberNotValid;
                %ABORT
            )
        )
    }
```

C.6 Seller updates an offer

Use case

UseCase: Seller updates an offer

Scope: Marketplace
SuD: Computer System
Primary actor: Seller

Main success scenario specification:
1 Seller locates a previously submitted offer.
2 Seller requests the system to update the offer, providing new details (e.g., price).
3 System responds with a request for the seller to prove identity.
4 Seller responds with the authorization number returned when the offer was submitted.
5 System validates the request and seller's identity.
6 System updates the offer.

Extensions:
4a Seller cannot provide the authorization number.
4a1 Use case is aborted.
5a Authorization number is not valid.
5a1 Use case aborted.

Procase

```
?Sl.locateSubmitOffer;
?Sl.requestUpdateOffer;
!Sl.requestProveIdentity;
?Sl.respondAuthorizationNumber;
(
    #sellerCannotProvideAuthorizationNumber;
    %ABORT
+
    #validateIdentity;
    (
```

```
            #updateOffer
        +
            #authorizationNumberNotValid;
            %ABORT
        )
    )
```

Procase after Preprocessing

```
?Sl.locateSubmitOffer {
};
?Sl.requestUpdateOffer {
    !Sl.requestProveIdentity
};
?Sl.respondAuthorizationNumber {
    (
        #sellerCannotProvideAuthorizationNumber;
        %ABORT
    +
        #validateIdentity;
        (
            #updateOffer
        +
            #authorizationNumberNotValid;
            %ABORT
        )
    )
}
```

C.7 Supervisor makes an internal audit

Use case

UseCase: Supervisor makes an internal audit

Scope: Marketplace
SuD: Computer System
Primary actor: Supervisor

Main success scenario specification:
1 Supervisor searches the database of offers for sensitive keywords in item description.
2 System displays the description of the item.
3 Supervisor removes the item from the database of currently visible offers.

Extensions:

1a Supervisor did not find any match.
1a1 Use case terminates.
2a Supervisor did not find any offending items.
2a1 Use case terminates.

Procase

```
?SU.searchDatabase;
(
    #doNotFindSupervisorMatch
+
    #displayDescription;
    (
        ?SU.removeOffer
    +
        #supervisorDoNotFindOffendingItem
    )
)
```

Procase after Preprocessing

```
?SU.searchDatabase {
    (
        #doNotFindSupervisorMatch
    +
        #displayDescription;
        (
            $RO
        +
            #supervisorDoNotFindOffendingItem
        )
    )
};
(RO:
    ?SU.removeOffer {
    }
+
    NULL
)
```

C.8 Seller to Clerk

Use case

UseCase: Seller to Clerk

Scope: Marketplace
SuD: Clerk
Primary actor: Seller
Supporting actor: Computer System

Main success scenario specification:
1 Seller submits item description to the clerk.
2 Clerk submits the description to the system.
3 Clerk reports the system response to the seller.
4 Seller submits the price, billing and contact information to the clerk.
5 Clerk enters the price, billing and contact information to the system.
6 Clerk reports the system response to the seller.

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

Procase

```
?Sl.submitItemDescription;
!CS.submitItemDescription;
(
    !Sl.reportSystemResponse;
    ?Sl.submitPriceBillingContactInformation;
    !CS.enterPriceBillingContactInformation;
    !Sl.reportSystemResponse
+
    #validationPerformSystemFail;
    %ABORT
)
```

Procase after Preprocessing

```
?Sl.submitItemDescription {
    !CS.submitItemDescription;
    (
        #validationPerformSystemFail;
        %ABORT
    +
        !Sl.reportSystemResponse;
        $SPBCI
    )
};
(SPBCI:
    ?Sl.submitPriceBillingContactInformation {
        !CS.enterPriceBillingContactInformation;
        !Sl.reportSystemResponse1
```

```
    }
+
    NULL
)
```

C.9 Buyer to Clerk

Use case

UseCase: Buyer to Clerk

Scope: Marketplace
SuD: Clerk
Primary actor: 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.

Procase

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

Procase after Preprocessing

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

C.10 Supervisor validates a seller

Use case

UseCase: Supervisor validates a seller

Scope: Marketplace
SuD: Supervisor
Primary actor: Computer System

Main success scenario specification:
1 Computer system asks the supervisor to decide on permitting a seller to operate on the marketplace.
2 Supervisor validates the seller and signals the system to permit the seller to operate.

Procase

```
?CS.decidePermitSeller;
#validateSeller
```

Procase after Preprocessing

```
?CS.decidePermitSeller {
    #validateSeller
}
```