Montpellier Academy
University of Montpellier II
Computer science department

Master Thesis in Computer Science

Conducted at The Montpellier Laboratory of Informatics, Robotics, and Micro-electronics

specialty : **Unified Research and Professional in Informatics**

Exception Handling from requirement specification to implementation : Extending UML

by Akram AJOULI

Defence on : **June 22, 2010**

supervisors : Christophe **Dony**
Clementine **Nebut**
Chouki **Tibermacine**

Preface

This report is the result of a master thesis research training prepared at The LIRMM(The Montpellier Laboratory of Informatics, Robotics, and Micro-electronics) in The Informatics department .This master thesis is also the last part of my Master of Science degree at University of Montpellier 2.

The Montpellier Laboratory of Informatics, Robotics, and Micro-electronics (LIRMM in French) is a cross-faculty research entity of the University of Montpellier 2 (UM2) and the National Center for Scientific Research (CNRS).

The Informatics department covers topics that range from the leading edge of modern mathematics to applied research: graph algorithms, bioinformatics, cryptography, networks, databases and information systems (data integration, data mining, coherency maintenance), software engineering (programming languages, objects, components, models), artificial intelligence (learning, constraints, knowledge representation, multi-agent systems), human-machine interaction (natural language, visualization, Web semantics and e-learning).

I would like to thank the following persons: Mr.Christophe DONY, Miss.Clementine NEBUT and Mr.Chouki TIBERMACINE for being my supervisors at LIRMM. Also thanks to Miss.Madalina CROITORU, Mr.Rodolphe GIROUDEAU and Mr. Abdelkader GOUAICH as being part of jury members.

Abstract

Exception handling is frequently considered as the final task to achieve when developing programs. However dependable systems need from their stakeholder to predict all exceptional situations, because any non-considered exceptional behavior potentially makes great damages. In our approach we have made an extension to UML2.0 which makes it possible to consider exception handling during applications modeling phase of the software life cycle . Our approach makes also opportunity to integrate the exception handling system explicitly in Use case, sequence diagrams allowing developer generating the last one, the class diagram and the aimed programming language code automatically with some needed refinements. Those automatic generations adapt the exception handling system specifically to each phase making some kind of communication between all phases of software life cycle. This can control any changes made on exceptional handling system by reflecting them on all phases of development. We have proceed to build our approach with defining a new UML2.0 extension profile and we have used model driven engineering to automatize diagrams and code generation making possible to specify exception handling and automate its passage from one phase to an other along the software life cycle.

Contents

List of Figures

1 Introduction

Since the failure of Flight 501[14] which is the first test flight of the European Ariane 5 on June 4, 1996, developers and stakeholders have been found obliged to put a great mark on exceptions and programs errors. Because when we hear that an integer overflow or an arithmetic overflow caused a loss of more than 370 million Us-dollar we must think about it seriously and we should ask if the developer of the software integrated on the plan we will ride,has made a consistent exception handling system. Exception handling is very important specially when we talk about dependent systems because many disasters were caused by non handled or non predicted exceptions. Designers, developers and software stakeholders should model exception handling system early when developing a software, because thinking about software exception on the last phase of software developing they will be found under pressure because they must focus on the normal and the exceptional behavior of the application in the same time, so any ignored exception may cause great disaster.

Exception handling has been always related to the implementation phase when developing softwares. Despite many programming languages support exception handling, developers have always found many problems in making a consistent exception handling system in the same time of implementing a software. In addition when we talk about dependent systems or critical embedded systems such as planes, we must be sure that thinking about exception handling in the last minute will be harmful and could cause damages to human lives if a developer forget to make a handler for an exception.
The objective of this master thesis is to study the exception handling system from the requirements specification phase to the final phase when developing a software. This study is based on extending UML by a profile which allows the designer and even the developer modeling the aimed exception handling system on the earliest phases of developing and then receiving automatically the traces of this system on the application code according to the target programming language.
This master thesis aims the following issues:

- Defining a UML profile in order to support exception handling system along all the software life cycle,
- Generating semi automatically the sequence diagrams from use case diagram,
- Generating automatically class diagrams,
- Generating application code,

- Moving automatically the exception handling system when passing from a diagram to an other until arriving to the generated code,
- Controlling any changes made on the exception handling system at any software life cycle phase and applying them to other phases,
- Specifying constraints that are based on matching the exception handling system to the programming language features in the phase of code generation.

Some propositions have been found on integrating exceptions handling along software life cycle, each of those studies was occupied by integrating exception handling notion on a one part of software life cycle and not on other phases.
Our vision to exception handling was more global because it covers all software development phases by exception handling notion. In addition our work offers many benefits to developers and designers by giving them opportunity to generate semi-automatically sequence diagrams with exceptions specified on use case diagram and full-automatically class diagrams and software code.
The rest of this document is structured as follows. Section 2 gives Background informations on exception handling and Unified Modeling Language (UML). Section 3 presents existing and related works. Section 4 describes our approach. Section 5 presents the implementation of our approach. Section 6 concludes and gives some perspectives.

2 Background

We will present in this section some backgrounds to the reader in order to put him in the right context. First, we give an idea about exception handling, then we present some UML diagrams which will be concerned by our approach.

2.1 Exception handling Overview

Exception handling is a programming language construct or computer hardware mechanism designed to handle the occurrence of exceptions, special conditions that change the normal flow of program execution.
Exception handling is the process of:

- Examining an exception message which has been issued as a result of a run-time error
- Optionally modifying the exception to show that it has been received (that is, handled)
- Optionally recovering from the exception by passing the exception information to a piece of code to take any necessary actions.

Exception definition: An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an exception.

Advantages of Exceptions: Using exceptions in programs has many advantages such as :

- Separating Error-Handling Code from "Regular" Code
- Propagating Errors Up the Call Stack
- Grouping and Differentiating Error Types

Exception handling mechanism After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The

list of methods is known as the call stack (see figure.1) [1].

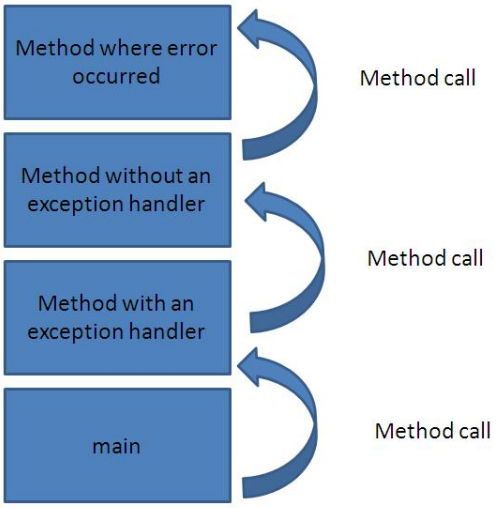

Figure 1: The call Stack

The runtime system searches the call stack for a method that contains a block of code that can handle the exception(see figure.2)[1]. This block of code is called an exception handler. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.
The exception handler chosen is said to catch the exception. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.

[1]Extracted from Sun Developer Network Site.

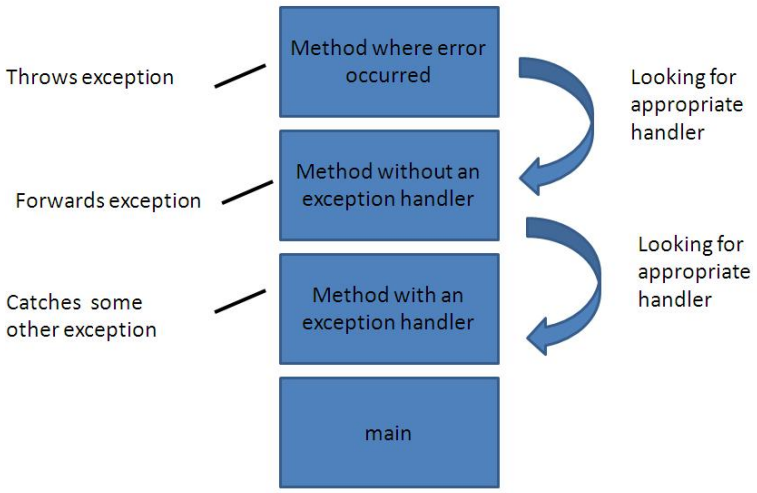

Figure 2: Searching the call stack for the exception handler

2.2 An overview of the used UML diagrams

UML includes a set of graphical notation techniques to create visual models of software-intensive systems. We will present in the following only the UML diagrams which we will use in our proposed approach.

Use case diagram A UML use case diagram is a type of behavioral diagram defined by and created from a Use-case analysis. It defines the system frontiers with the actors that interact with the system and it defines in the outside the set of the system functionalities structured by use case which are eventually related to each other with inclusion, extension or generalization.
The main purpose of a use case diagram is to show what system functions are offered to actor. Roles of the actors in the system can be depicted.

Sequence diagram A UML sequence diagram is a kind of interaction diagram that shows the dynamic side of the system.

Class diagram A UML class diagram represents the static structure that describes the structure of a system by showing the system's classes, their features, and the relationships between the classes.

3 State of the art

While a large part of the exception handling research community has worked on exception handling mechanism at code level, a small part of them has tackled more globally how exceptions should be dealt with all over the software life cycle. We present these existent works in the following subsections.

3.1 Classifying exceptions according to software lifecycle

Romanovsky et al.[6][13] classify exceptions into three levels: exceptions related to application, exceptions related to design and exceptions related to implementation. In each phase of a software life cycle, developers should discover its related exceptions and define right handlers. Handlers defined in each phase should handle exceptions related to the same phase and those related to previous phases. This work was done to resolve the problem of loosing the exception raising context, it aims to define exceptions and handlers in the context in which errors occur.

3.2 Exception handling in requirement specification

3.2.1 Exceptional use case

Kienzel et al. [2] have presented exceptional use case. Their idea consists on extending the UML in order to adapt use case to exceptional behaviors. Extensions affect the use case itself by presenting a new approach of use case which consists on defining a use case as a handler. So, we can find normal or standard use case and exceptional use case. The figure.3 represents a use case diagram made for an elevator system [2]. We see in this diagram the use cases stereotyped as *handlers* in order to distinguish them from normal use cases, we see also comments stereotyped as *Exception* in order to show the name of the exception associated with a given use case, we can see also links stereotyped as *interrupt and continue*, others are stereotyped as *interrupt and fail*. These two last stereotypes are defined to express in the use case diagram the model of exception handling model such as resume and termination. Kienzel et al. also tackled the problem of exceptions and handlers documentation. Further works [3][4][5] propose a tool which simulates the safety and reliability of a dependent system using use cases refinements. This tool allows developers to simulate the reliability of a dependent system by proceeding iteratively and discovering exceptions until getting the demanded level of reliability and safety.

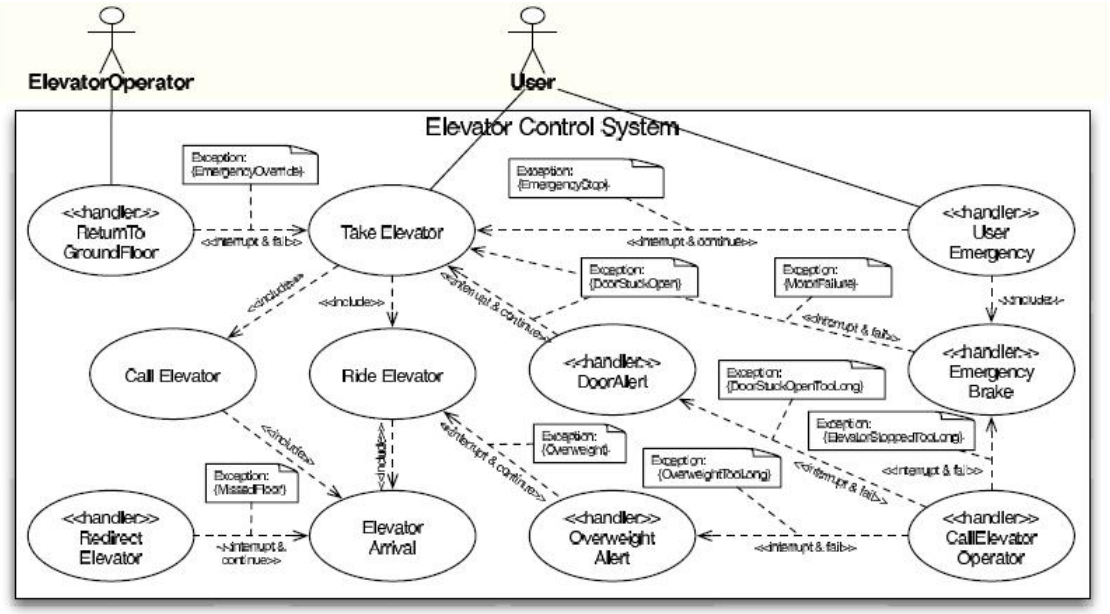

Figure 3: Exceptional Use Cases (extracted from [2])

3.2.2 Misuse Cases

Alexander Ian [12] proposes a new kind of use cases called Misuse cases. A misuse case is a use case that focuses on non-functional requirements.

In [12] a scenario is a sequence of actions leading to a Goal desired by a person or organization. On the other hand, a negative scenario or a misuse case is a scenario whose Goal is desired not to occur by the organization in question and desired by a hostile agent (not necessarily human)

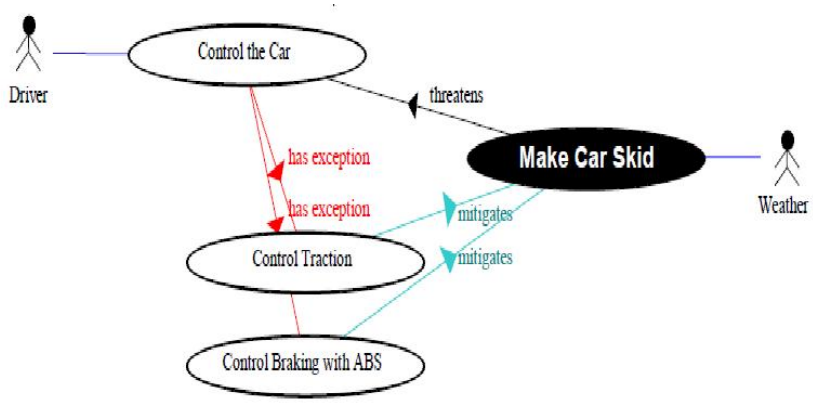

Figure 4: Misuse Cases Approach (extracted from [12])

Figure.4, extracted from [12], shows a Driver that participates as an actor in the use case (control the car). The weather is an other actor, but not an ordinary one because it participates in a misuse case (make car skid), that threatens the normal use case and results two exceptions represented by two use cases (control traction and control Braking with ABS).

3.3 Exception Handling in sequence diagram

Oddleif Halvorsen et al. [8] present the notion of time exceptions in sequence diagrams. The authors point out that the UML does not explicitly describe time exceptions, they augmented the UML by showing in sequence diagrams exceptions triggered by the violation of time constraints. They give also a formal definition of time exception in sequence diagrams. In order to explain their approach they used an ATM example.

Figure.5 describes the normal scenario of a normal withdrawal sequence diagram. The user is expected to insert the card and put four digit numbers, then while the ATM verifies with the bank if the code entered is correct, it

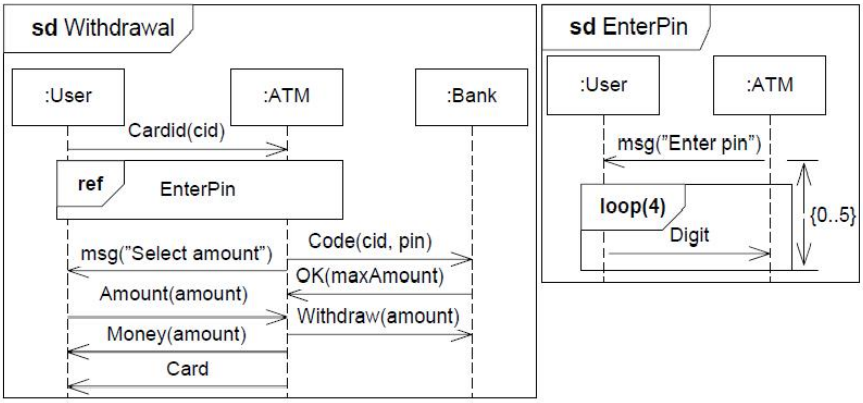

Figure 5: Normal scenario of withdrawal (extracted from [8])

asks the user to enter the value of demanded amount, finally the user gets the amount. Figure.6 describes an exceptional scenario for a withdraw and how time exception was applied to sequence diagram. Because in ATM there is constraint on time reserved to enter the four digit, so, if time is over and the four digit are not received by the ATM, there will be a time exception triggering which must be handled. In order to put the user card in a safe place, supposing that user left the ATM and forgot his card, and in order to serve the next user by canceling the previous service, those details can be clearly seen in the figure.6. In this figure UserLeftCard sequence diagram represents the handler of time exception UserLeftCard.

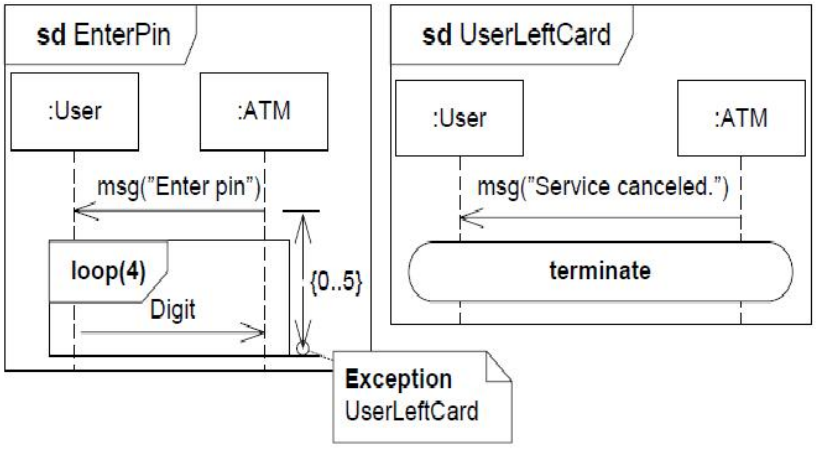

Figure 6: Applying Time exception (extracted from [8])

3.4 Modeling exception handling by extending modeling languages and extending aspect oriented language

Alessandro Garcia et al. [9] present the problem of absence of explicit specification of exception handling both in modeling languages and in development environments.
Their approach is oriented to model driven software development (MDSD). They started by demonstrating that even we could express exception handling in the state machine diagram it will be very complex and it is better to make a separate model for exception handling.

3.5 UML and exception handling:

UML does not fully describe exception handling: exceptions are mentioned for activity diagram and in method specification.

3.5.1 Existence of exception handling in Activity diagram

The try block can be modelled by a protected node in an activity diagram. The catch block is called the handler body (see figure.7). If an exception occurs, the set of handlers is examined for a possible match . If a match is found, the handler body is invoked and the handler catches the exception. If the exception is not caught, the exception is propagated to the enclosing protected node if one exists.(see figure.7)

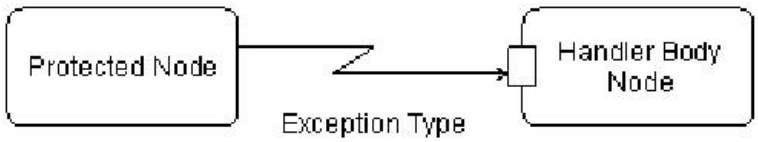

Figure 7: The UML 2.0 Exception Handling Notation (extracted from [UML Superstructure])

Let us look at the following example given in [16] to understand the notation of exception handling in the activity diagram.(see figure.8)
Consider a simple URL viewer that prompts the user for a URL and then copies the contents to the display. The logic for this activity is to open a new URL, open the display, copy the contents of the URL to the display, and then close the streams for the URL and the Display. In the sunny day case, we simply display the contents of a resource.

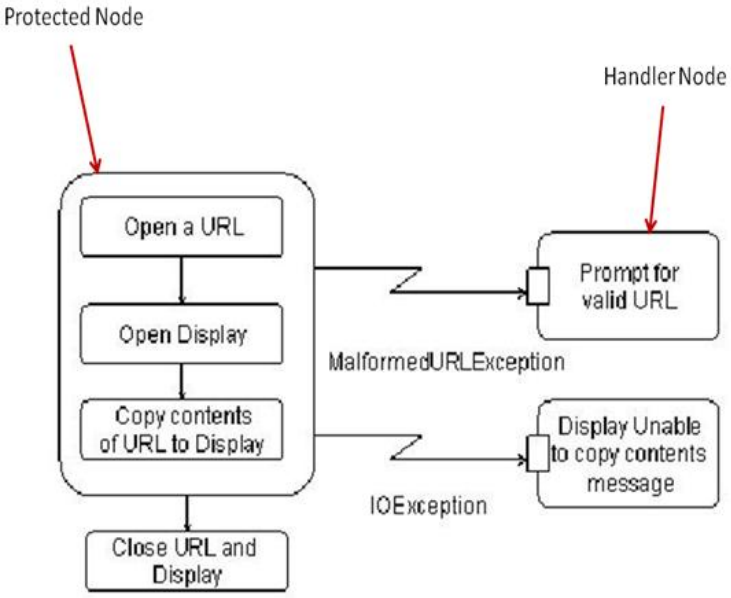

Figure 8: The URL viewer example (extracted from [UML Superstructure])

During the course of this logic, two possible exceptions may result. The first is a MalformedURLException. This exception could be detected when we open the URL. The second is an IOException that could happen in several places. The URL might be valid but the object that it locates is unreadable. The display may not be accessible.
In the model of this system, there are four activities. Three of these activities are nested in a protected node. In UML 2.0, we are allowed to nest activities. There are two handler bodies; one for the MalformedURLException and one for the IOException. Successors to the handler bodies are the same as the successors to the protected node. We could, therefore, look at the activity which closes the URL and the Display as our "finally" clause.

3.5.2 Exceptions in UML classes

In the specification of behavior features made by OMG, exceptions can be associated to behavior features such as operations which could raise these exceptions.

Even UML offer a small supporting of exception handling, it does not permit to model it and to show it explicitly in the most important diagrams of softwares life cycle such as use case diagrams, sequence diagrams and class diagrams.

4 Proposed exception handling model and UML extensions

The study of the literature concerning exception handling in early design phases led us to propose our approach materialized by a UML profile, to led the designer consider exceptions from refinement analysis to implementation.

4.1 The proposed UML Profile

According to [7] a profile in the Unified Modeling Language (UML) provides a generic extension mechanism for customizing UML models for particular domains and platforms. Extension mechanisms allow refining standard semantics in strictly additive manner, so that they can not contradict standard semantics.

Profiles are defined using stereotypes, tag definitions, and constraints that are applied to specific model elements, such as Classes, Attributes, Operations, and Activities. A Profile is a collection of such extensions that collectively customize UML for a particular domain (e.g., aerospace, healthcare, financial) or platform (J2EE, .NET).
In order, to adapt UML to our approach we have defined a profile which contains many stereotypes that extend the UML meta classes to be able to show explicitly exception handling notations in the aimed locations. In the following we will present firstly the whole profile, then we will describe each stereotype separately.
Figure.9 shows the proposed profile. The description of this profile and its defined stereotypes will be presented in the following two sections.

4.2 Proposed Modeling Process

Our approach is based on seven main steps(see figure.10) which will be described in the following:

4.2.1 Step1: Defining use case diagram and discovering exceptional use case

In this step the designer has to define the use case diagram of the target application, then he should look for the use case in which there could be raised Exceptions. Then he has to extract the set of exceptions that could be raised with each exceptional use case. To distinguish graphically normal use case and exceptional use cases we propose an extension to UML in order to support notating exceptions in the use case diagram.
This step is very important, because as you now use case diagrams are made to define requirements when developing softwares. With the standard UML,

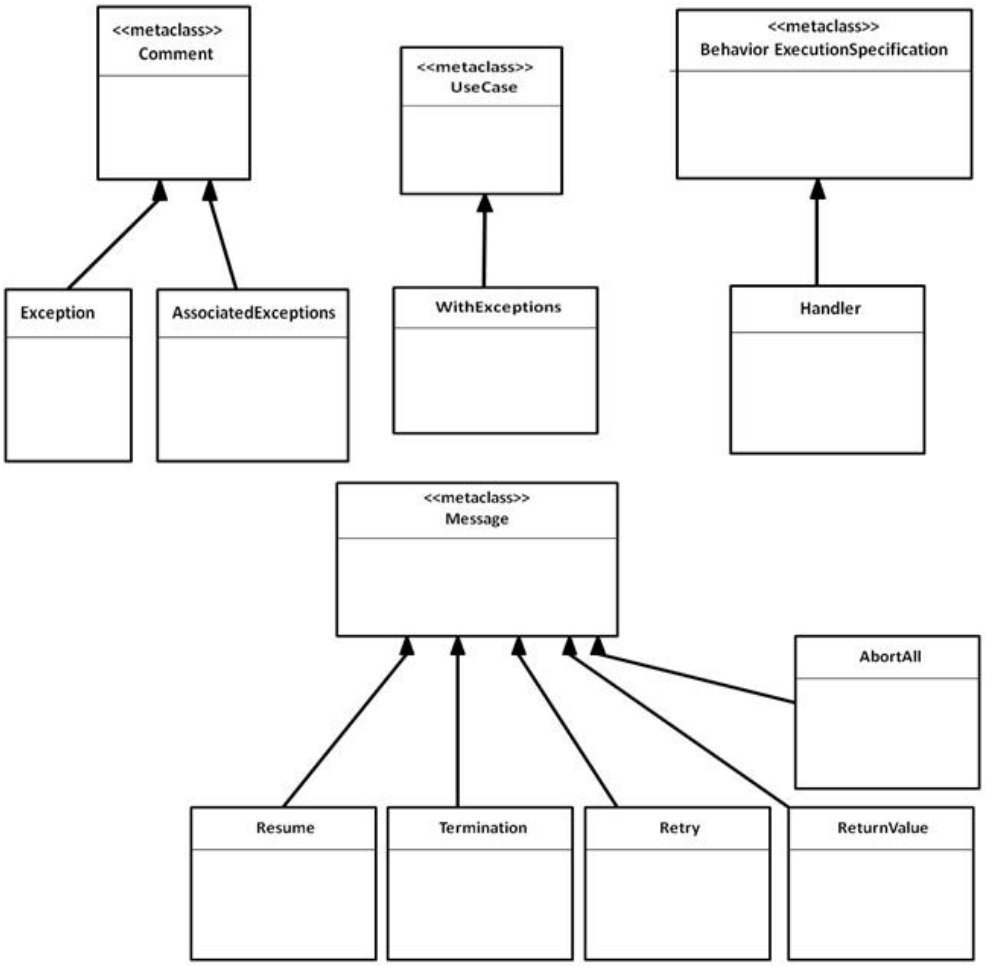

Figure 9: Exception Handling Profile

designers could only define normal use case or normal behavior, but we see that a use case diagram will be more useful if it includes also an explicit or a graphic notation that permits to separate normal behavior from exceptional behavior.
This step could be iterative, and it is more efficient if it will be, because in dependent systems there is not only one person who is occupied by defining requirements, but there are stakeholders also, and the set of requirements must be verified times and times before passing to other software life cycles phases, therefore designers could discover new exceptions each time there is a requirement verification. The exception notations in use case diagrams are now possible with our proposed extension to UML.

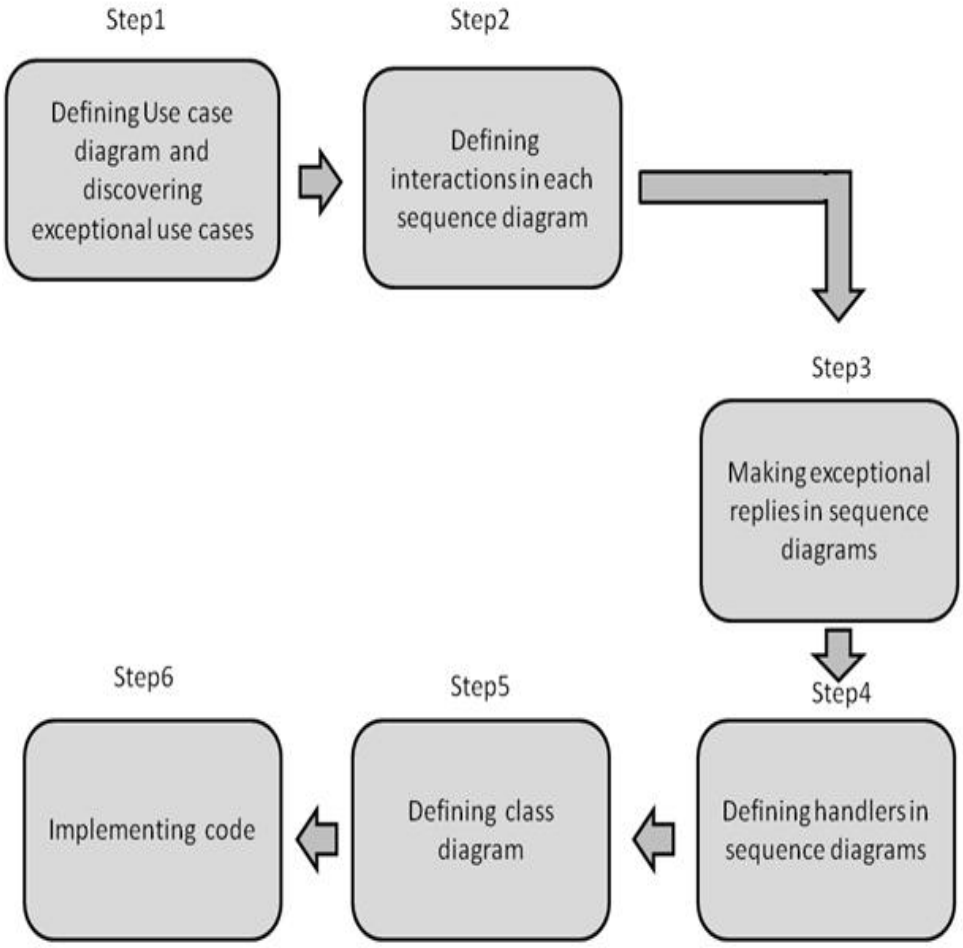

Figure 10: Process of our approach

UML extension and proposed stereotypes for use case diagrams
We will present in the following each stereotype separately:

i) WithExceptions stereotype This stereotype extends the meta class Use Case. Therefore it is defined to be applied to use cases. Its application to a use case indicates that this one contains exceptions. This characteristic gives to the designer the opportunity to distinguish between normal behavior and exceptional behavior explicitly on the use case diagram.
This stereotype is also necessary for implementing our approach because it permits performing use cases classification. Applying this stereotype to use cases which contain exceptions will give an identification attached to those exceptional use cases in order to distinguish them from normal use cases. The use case diagram will so show explicitly two behaviors: one normal

and other exceptional. This is very important, specially when developing dependent systems, because designers and stakeholders will be more aware to exceptional use cases in order to avoid the worst.

This stereotype is to be applied to use case diagram. Figure.11 shows how an use case that contains exceptions looks like when we apply this stereotype.

ii) AssociatedExceptions stereotype This stereotype extends the meta class Comment, so it is defined to be applied to comments. It is used to show the set of exceptions associated to an exceptional use case. The designer can introduce the names of associated exceptions in the body of the comment stereotyped by ≪AssociatedExceptions≫ in order to give to this comment the specificity that allows the designer to extract carefully the set of exceptions that can be raised in the scenarios of an exceptional use case stereotyped by ≪ WithExceptions ≫ .
The figure.11 shows an example of a comment stereotyped by ≪ AssociattedExceptions ≫ and How use case diagram looks like with the application of this stereotype.
The other example is the example of the travel agency. In this example there are three main actors: the client, the broker and the provider. Each of these actors is concerned with many tasks. The figure.12 shows the use case diagram of the travel agency case study after applying our proposed UML extensions. The use case diagram shown by figure.12 demonstrates that with our proposed profile the designer can easily distinguish between normal behavior and exceptional behavior.

4.2.2 Step2: Defining interactions in each sequence diagram

A sequence diagram represents the set of interactions that describe a part of a use case. That is why the designer is demanded to define the necessary interactions that describe the use cases scenarios. The designer could do this work easier if he finds sequence diagrams skeletons already generated from the use case diagram. in this case, he will be asked just to complete the generated sequence diagrams skeletons.

4.2.3 Step3: Making exceptional replies in sequence diagrams

We mentioned in the description of the step1 that there are in the use case diagram exceptional use cases and normal use cases. So, each sequence dia-

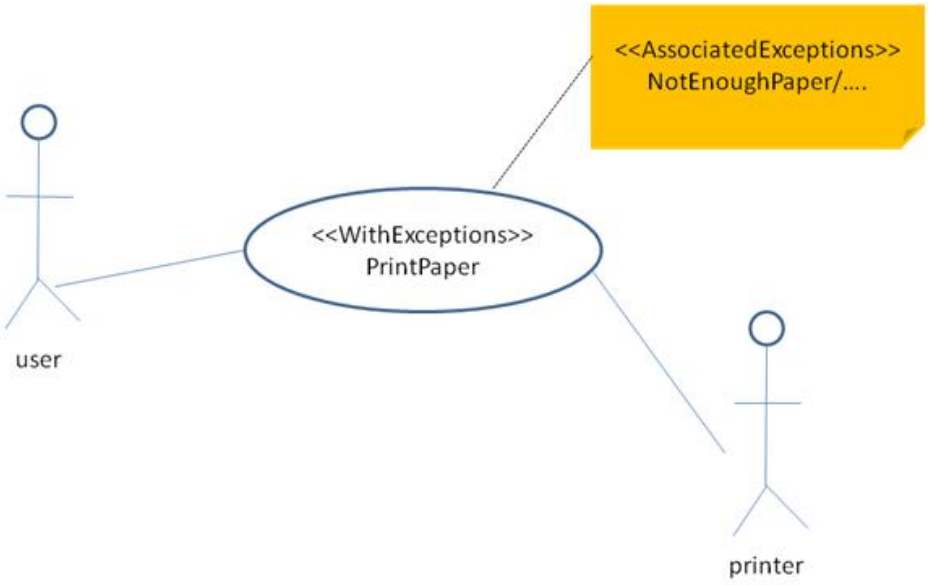

Figure 11: Example of use case diagram after applying new stereotypes.

gram that describes an exceptional use case will contain the set of exceptions that are associated to this use case. those exceptions are included in the sequence diagram as comments which will notate exceptional replies. A message which represents a response in the sequence diagram is a reply. So, we will find in the sequence diagram exceptional replies annotated by comments representing exceptions and normal replies. Therefore, a message which represents a method invocation will invoke two responses if the method could raise an exception. The relation between these two responses is exclusive, it means that either there will be a normal response or an exceptional reply that represents an exception raising.

UML extension and proposed stereotypes for sequence diagrams to express exceptions

Exception stereotype This stereotype extends the meta class Comment . It is defined in order to be applied to a comment that indicates in the sequence diagram that a message which is annotated by this comment represents an exception. Its use is very important in the sequence diagram because it provides a real and explicit view for the designer to distinguish between Normal interactions and exceptional interactions.
As each use case will be described by one or more sequence diagram, in the case of an exceptional use case we will find in the sequence diagram its describing interactions inheriting its associated exceptions included in the comment stereotyped by ≪ AssociatedExceptions ≫ . So, each comment

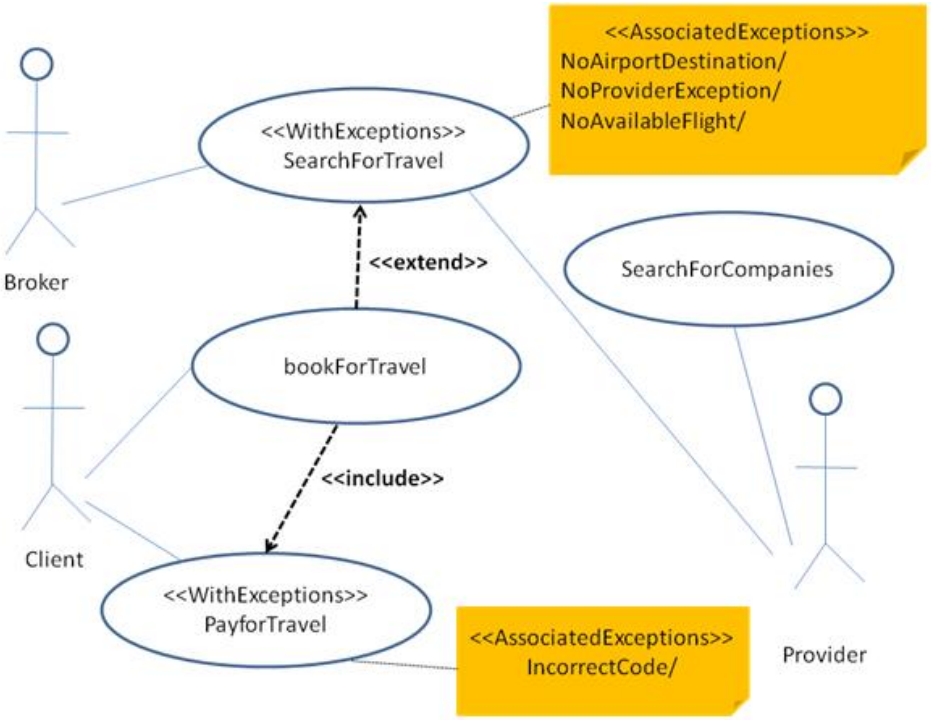

Figure 12: The travel agency use case diagram after applying the proposed stereotypes

stereotyped by ≪ Exception ≫ and owned by the sequence diagram will contain an exception name from associated exception. We think that sometimes images are more expressive than words . We invite you to see figure.13 to understand the whole story of the stereotype ≪ Exception ≫ .
Figure.13 shows an example of a comment stereotyped by ≪ Exception≫ and How sequence diagram looks like with the application of this stereotype. In this figure we present the example of sequence diagram that describes part of interactions between a user and a printer. After a request made by a user to print n pages, the printer will either reply with a normal response or an exceptional response. In the first case, it will satisfy the request of the user, in the second case it will raise an exception if there is not enough paper. You should note that relation between normal response and exceptional response is exclusive, it means that one of them will exclude the other according to the kind of the reply. You should note also that here we present only the utility of exception stereotype and we will present how to handle exceptions in sequence diagram in the subsection of handler specification.

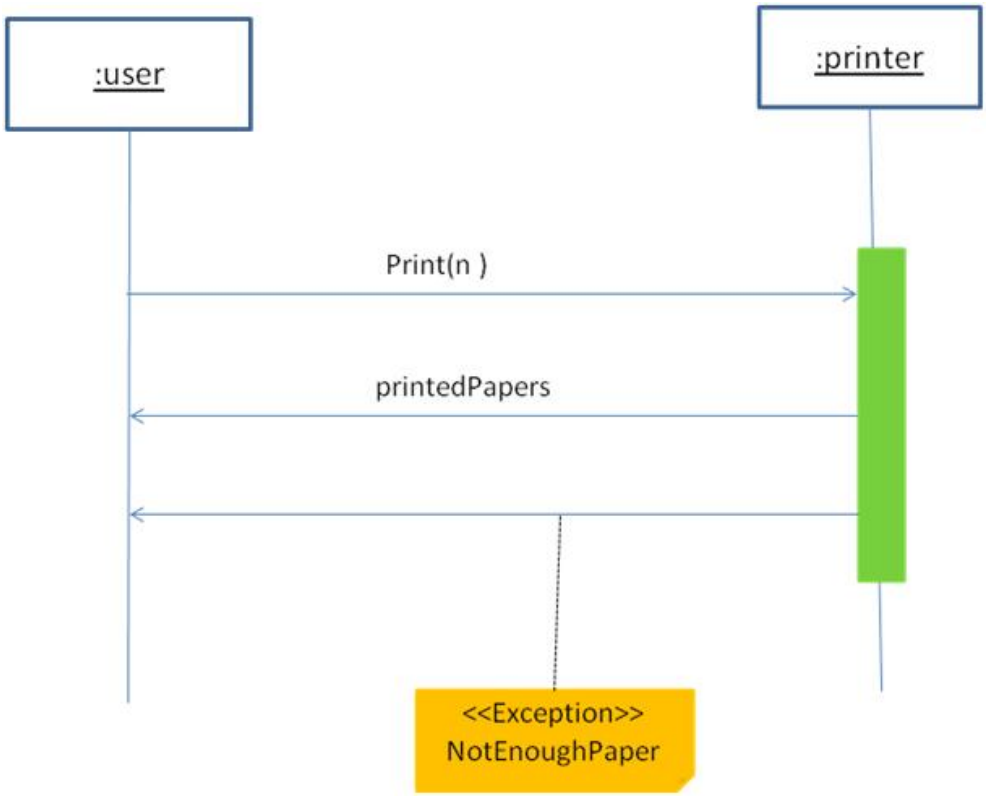

Figure 13: Example of sequence diagram after applying Exception stereotype

4.2.4 Step4: Defining handlers in sequence diagrams

After notating exceptional replies in the sequence diagram, the designer has to look for how defining handlers to the specified exceptions. To do this, the designer must make a handler to each exceptional reply. You should note that our proposed extensions to UML provide for the designer graphic notations to define handlers and manipulate graphically handling mode.
This step is more important than the previous step, because designer must make attention when defining handlers. This is evident, because an exception that is not handled correctly or efficiently will cause software failure.

UML extension and proposed stereotypes for sequence diagrams to express handlers

Handler stereotype This stereotype extends the meta class Behavior Execution Specification. Therefore it is defined to be applied to Behavior Execution Specification. Its application to a Behavior Execution Specification indicates the starting of handling an occurred exception. In the se-

quence diagram, exceptions are represented by an exceptional reply as we saw above. This reply is received by The invoker that called the method which raised the exception. It is evidently so, to put the handler in the life line that represents the invoker. With this stereotype, the designer could define handlers early, easily and graphically without any dependency to programming languages. This extension allows also maintaining softwares code easily, because if any changes of exception handling system are needed, they will be done directly from the sequence diagram, and then they will be reflected directly to the generated code. In order to show how to represent graphically a handler in the sequence diagram, we have so presented again the example of the printer(see figure.17) in order to show how to express a handler by graphic notation.

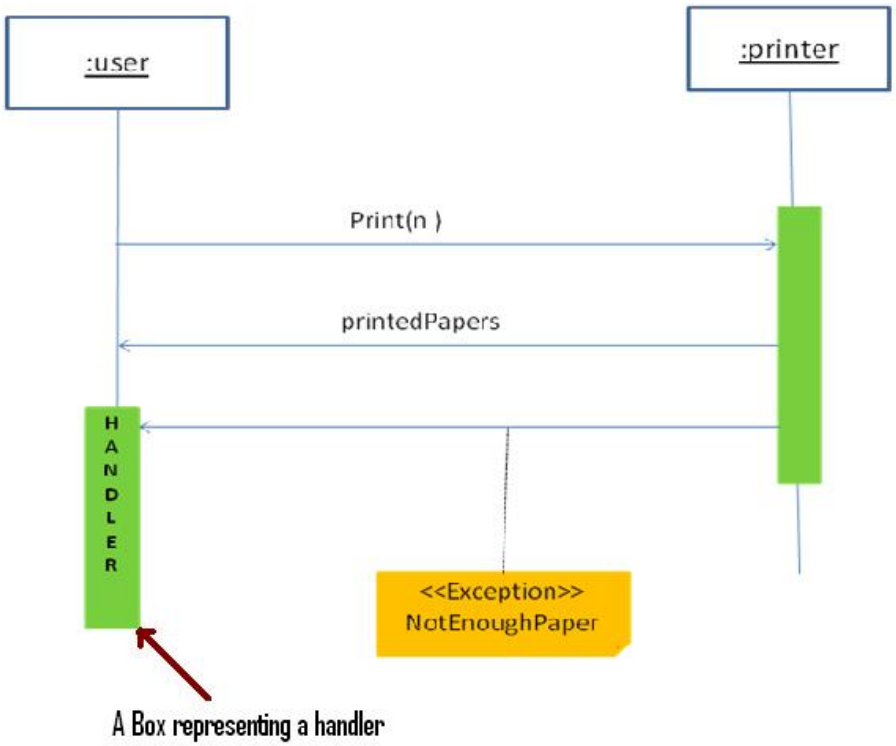

Figure 14: Use of Handler Stereotype

Handling an exception may be done in different modes which can be :

- Termination,
- Resume,
- Retry,
- Propagation,
- ReturnValue,
- AbortAll.

We consider these modes as constraints that will force the designer to respect them. These constraints will be tested specially in the generation code task when the designer or the developer chooses the target programming language. For example, suppose that developer chooses the resume mode when defining one or more handlers and then he chooses JAVA as target programming language, there will be an exception saying that the chosen programming language does not support resumption model [20], so you have to change the chosen handling mode or the target language programming.
In order to satisfy these handling modes we have proposed five stereotypes as extensions to UML. In the following, we explain how we have done these extensions and which UML meta classes we have extended.

Termination stereotype This stereotype extends the meta class Message(see figure.9). It indicates the excecution termination of the protected block in which the exception has occured. This extension to UML allows to the designer assigning the termination mode of an exception handling. It permits showing explicitly and graphically that a given handler will express termination handling mode. When developing dependent systems the graphic notation that indicates that a handler will cause software execution termination will be under discussion between designers and stakeholders to predict early if termination is useful or harmful.
To more understand this point let us take an example, suppose that we are filling a web application that contains 5 tasks, and in the final task there is an exception raising, in this case if the developer of the application made the termination mode to handle this exception we will loose all the informations which we have filled in the four first tasks. But if this case was studied early such as in the sequence diagram, designers will easily find the wrong way of handling such exception by distinguishing easily the graphic notation of the termination mode.
Its graphic notation is shown in figure.15. Termination mode represents a feature of many programming languages. This feature is implemented by the programming languages developers and it is not never clear and explicit to a programming language user. It was been always behind a black box code that is specific for company or persons who developed the programming

language. And that is why we want to make this handling mode graphic and explicit. We see that giving the designer opportunity to express termination in the sequence diagram is very useful, because with a graphic notation the designer will easily manipulate this handling mode without any dependency to programming languages features. Let us see a simple example that shows the usefulness of Termination stereotype and its graphic notation(see figure.16): a client can request an amount from a bank cash machine. This last one can reply with a normal response (receivedAmount), or an exceptional response (BadCredit exception). With termination mode execution will be returned in the next method which is out of the protected block. In this case, execution will start with seeBalance method invocation.

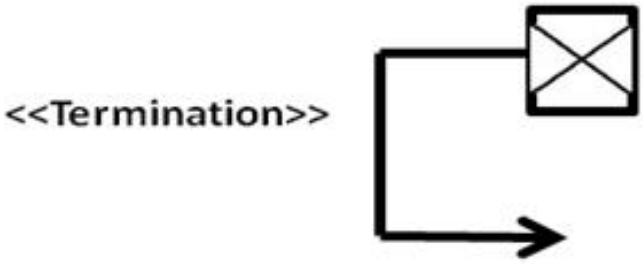

Figure 15: Graphic notation of Termination stereotype

In order to understand the interest of our approach and the usefulness of the proposed stereotypes, we will present with each handling mode a pseudo code that is supposed to be generated. Most of the presented pseudo codes are written with JAVA language. The only one which is not written with JAVA is the one that describes the resume mode (see listing.2) and which is written with SmallTalk, because JAVA does not support resume handling mode. The matching between the handling mode and the target programming language features will be considered as a constraint that will verify the conformance between the handling mode and the chosen programming language.
The pseudo code seen in the Listing.1 describes the termination handling mode shown in the figure.16 and how exceptions and handlers will be placed in the right try/catch blocs.

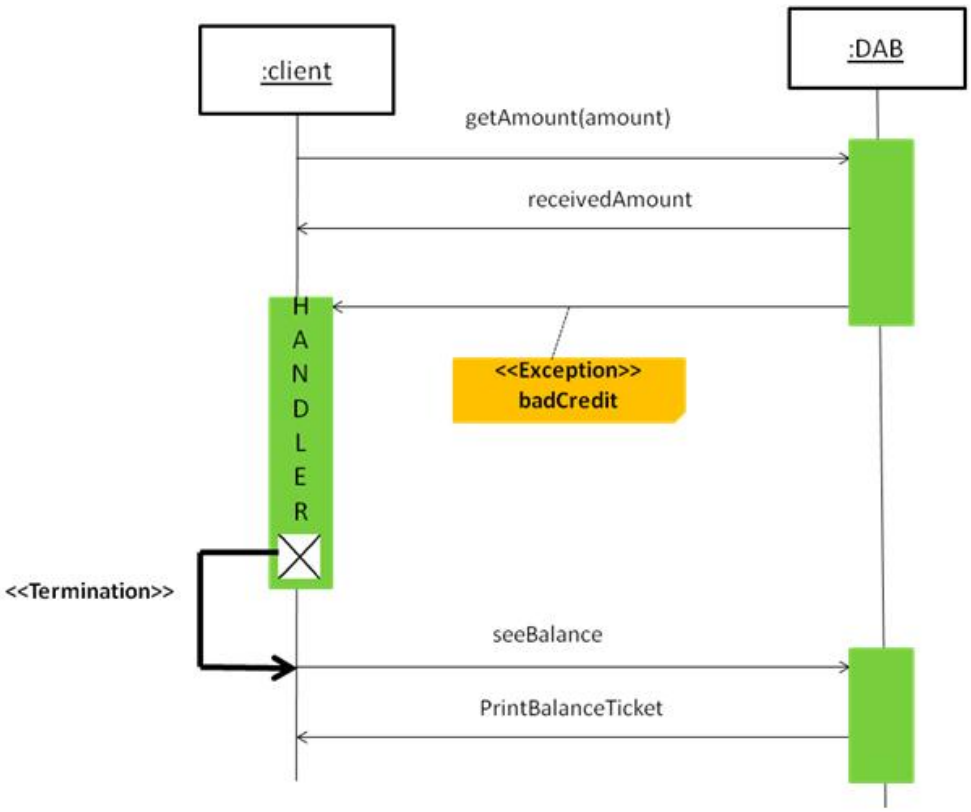

Figure 16: Application of termination handling mode

Resume stereotype This stereotype extends the meta class Message(see figure.9). Even programming languages that support resume mode have already its specific implementation that allows saving the point from which the exception was raised, but we have added this graphic notation in order to indicate exception handling mode of re-putting the system in the raising context again.
Its graphic notation is shown in figure.17. In order to explain the use of this stereotype and its graphic representation we have make a small sequence diagram that represents some interactions between a printer and a user. Those interactions can raise an exception named NotEnoughPaper which can be raised with PrintRequest method invocation. If this is supposed to happen the designer must put a box handler in the lifeline representing the user, then he must put the arrow representing the resume mode. Its base must be put in the box handler and its head must be put exactly in the place where the exception has been raised(see figure.19). With this notation the designer could easily express explicitly and graphically the resume mode of exception handling.

Listing 1: Termination handling mode

```
1
2   Class DAB{
3           .....
4           .....
5           .....
6           public void getAmount(int amount) throws BadCreditException{....} }
7
8   Class Client{
9            .....
10          DAB d = new DAB();
11          public void withdrawmoney(amount){
12          try{ d.getAmount(amount);}
13          catch(BadCreditException e){ System.out.println(e.getMessage());}
14          this.seeBalance();
15           }
16           }
```

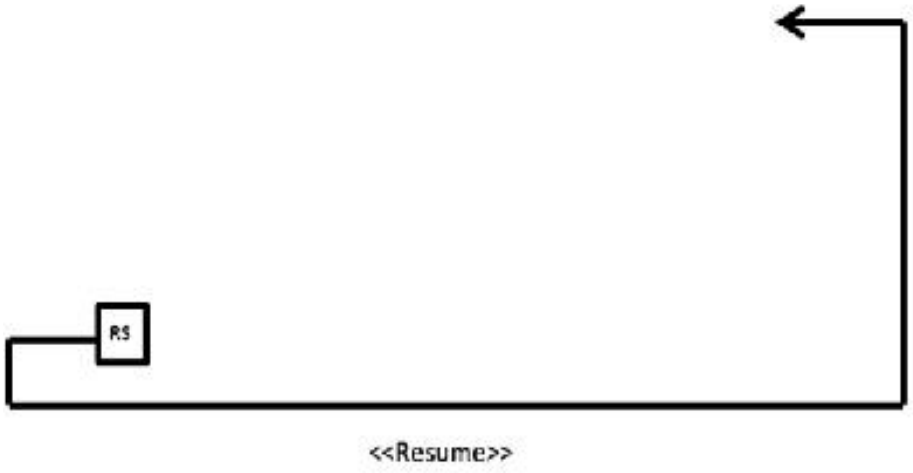

Figure 17: Graphic notation of Resume stereotype

The pseudo code seen in the Listing.2 describes the resume handling mode shown in the figure.17. This code is written with Smaltalk in order to show the resume handling mode that is specified in the sequence diagram shown in the figure.17.

Retry stereotype This stereotype extends the meta class Message(see figure.9). It represents the retry mode of exception handling. It allows the designer to express graphically that a given handler obliges a software user

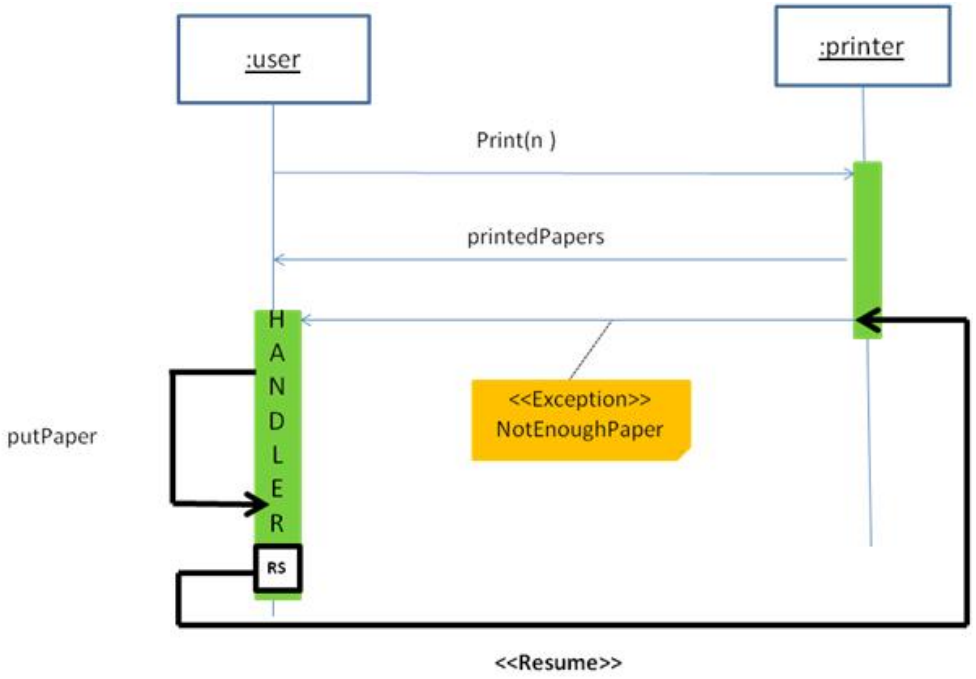

Figure 18: Application of resume handling mode

to modify some parameters after the occurrence of an exception and then re-execute the software.

This is very important in dependant systems, because in some urgent situations the system must be included in an interactive mode with the user after an exception was occurred in order to be re-executed with new corrective parameters.
Its graphic notation is shown in figure.19. To express the retry mode the designer must put the base of the arrow representing the retry mode and put its head in the place from which the message was sent to invoke the method that raised the exception. To clarify this idea let us take a look to the travel agency application (see figure.20.). As you see in this figure.20, a client enter a bank card code to book for a travel, he will receive either a normal response or an exceptional response. In this last case(Incorrect-code), the chosen handling mode retry will allow the client to enter the right code and then re-execute the method EnterCode(code).

Listing 2: The resume handling mode

```
1
2   Object subclass: #Printer
3     .....
4     .....
5     .....
6     Printer >> print....
7
8   Object subclass: #User
9      .....
10     .....
11     User >> PrintRequest
12     [ Printer print ]
13     with: NotEnoughPaper
14     do:[:e | show: ''put paper'' e resume]
```

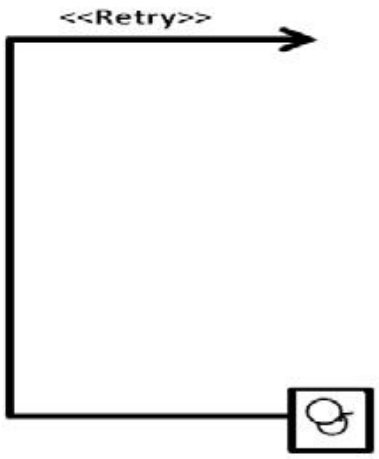

Figure 19: Graphic notation of Retry stereotype

The pseudo code seen in the Listing.3 represents the code that is is supposed to be generated and shows the retry handling mode specified in the sequence diagram shown in the figure.20.

SignalException Sometimes a handler is defined just to signal the same exception which has to handle or a new exception. This functionality is offered by all programming languages that support exception handling system. Every language express signaling exception by its own key word(like throw in JAVA, signal in Smalltalk...etc). Its graphic notation is the same as the one that express an exception in a sequence diagram(see figure.13). Let us understand how to express re-signaling an exception by a handler

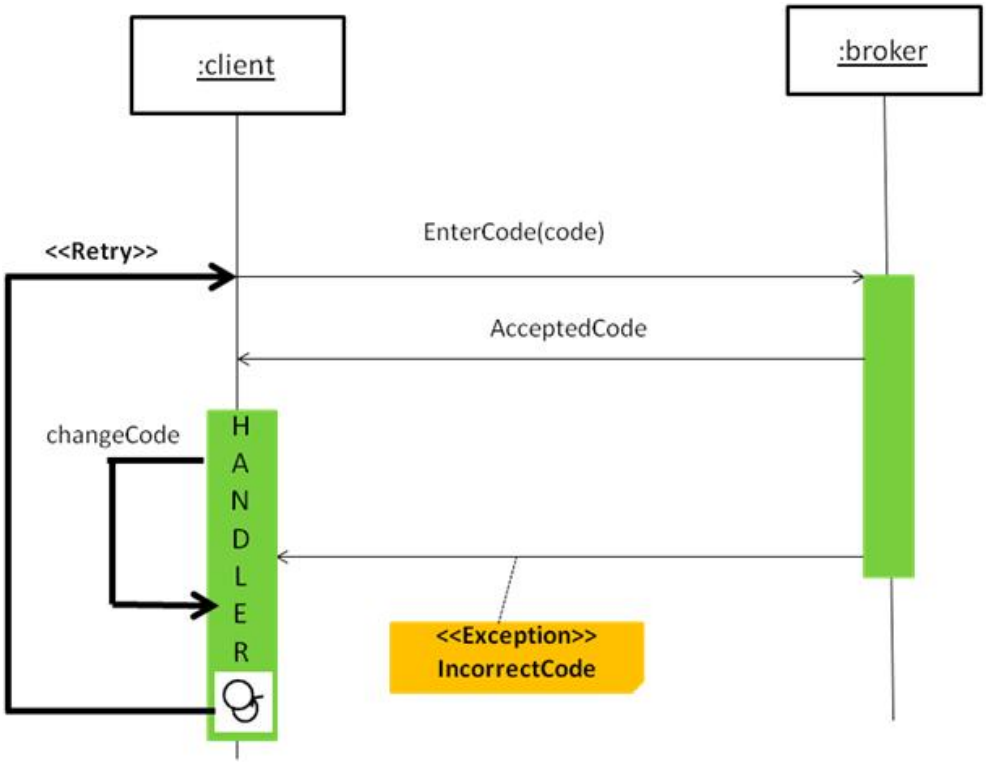

Figure 20: Application of retry handling mode

by looking again to the travel agency example. As you see in the figure.21, the provider can signal a BadParameterException, if so the the handler defined in the Broker will signal a new exception(NoAirportDestination) to the client which must also handle this exception. This example shows the importance of expressing exception propagation graphically, because the designer can control the flows of exceptions easily. In addition the sequence diagram gives a clear idea about the progressing of method invocations, and this helps the designer to understand easily the flows of exceptions if there are successive methods invocations accompanied by many raised exceptions.

The pseudo code seen in the Listing.4 presents how the propagation handling mode specified in the sequence diagram shown by the figure.21 looks like.

ReturnValue stereotype As an exception could be handled by returning a value to the invoker of the method that raises the exception we have defined the stereotype ≪ Returnvalue ≫ in order to allow to the designer expressing this handling mode and finding a graphic notation which

Listing 3: The retry handling mode

```
1
2   Class Broker{
3         .....
4         .....
5         .....
6         public void enterCode(int code) throws IncorrectCodeException{..} }
7
8       Class Client{
9            .....
10           Broker b = new Broker();
11           public void pay(int cardcode){
12           boolean accepted = false;
13           while(!accepted){
14           try{ b.enterCode(cardCode);
15                accepted = true;}
16           catch(IncorrectCodeException e){
17           System.out.println("Invalid code, retry again with the right card code");}
18                  } }
19                  }
```

Listing 4: The resignaling handling mode

```
1
2
3
4
5  Class Provider{
6    .....
7    .....
8    .....
9    public void lookForDestination(String destination) throws BadParameterException{..}
10
11 Class Broker{
12      .....
13      Provider p = new Provider();
14      public void findDestination(selecteddestination) throws NoAirportDestination{
15
16      try{ p.lookForDestination(selecteddestination);}
17      catch(BadParameterException e){ throw new NoAirportDestination;}
18              } }
```

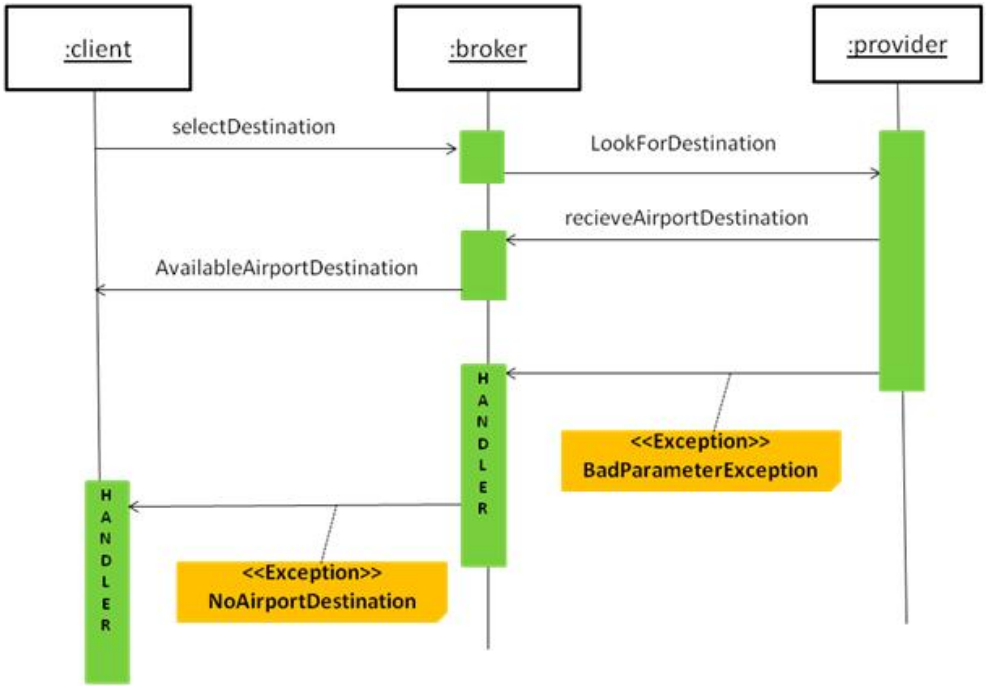

Figure 21: Application of resignaling handling mode

could be manipulated easily. When using a message stereotyped by ≪ Returnvalue ≫ the designer has to enter as parameter a value.
Its graphic notation is shown in figure22. The figure.23 shows a standard example of interactions between a client and a bank cash machine supposing here that the client could be classified in a bad list of clients who have bad history with their bank. If this is the case we see in figure.23 how we could express the return value handling mode. In the example shown in this figure, we see that the handler put to handle the ClientinbadList exception swallows the bank card and return a string value to the client. This example shows how with a simple arrow the designer can easily express return value in exception handling.

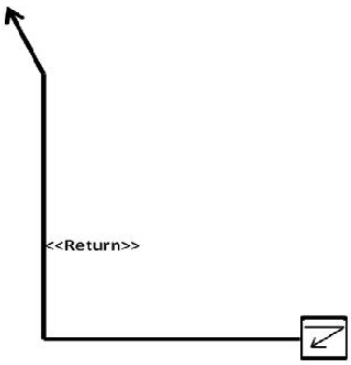

Figure 22: Graphic notation of ReturnValue stereotype

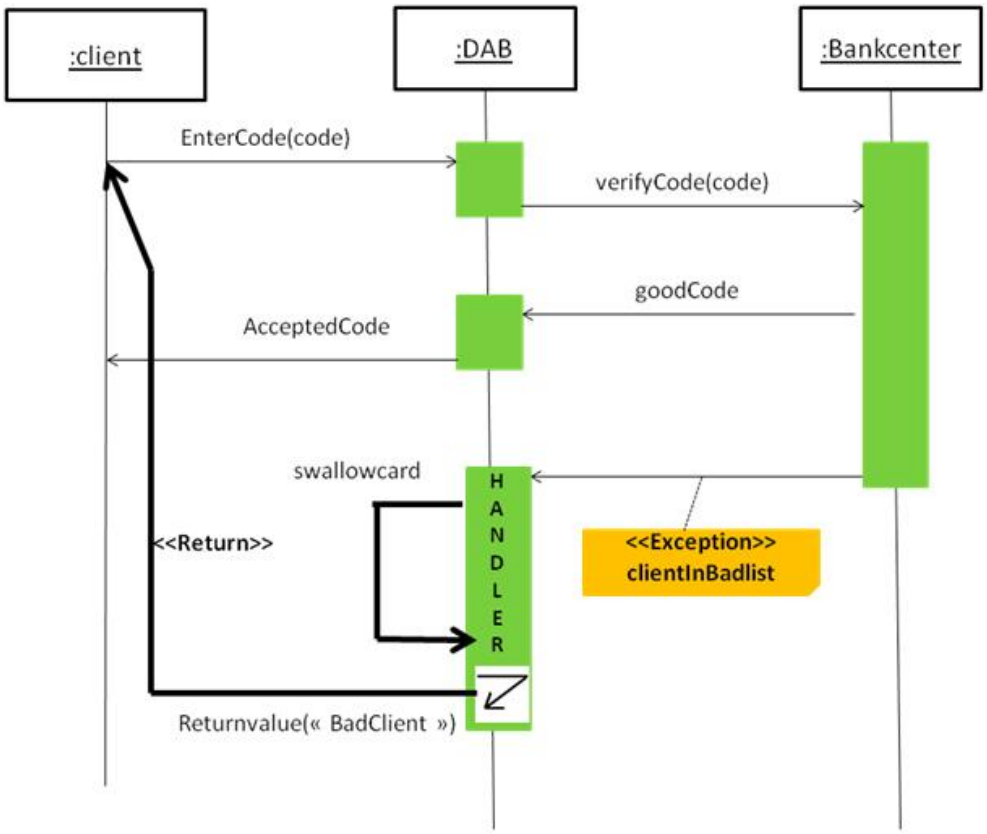

Figure 23: Application of return handling mode

Listing 5: The returnValue handling mode

```
1
2
3
4
5     Class Bankcenter{
6           .....
7           .....
8           .....
9           public void verifyCode(int code) throws ClientinBadlistException{....} }
10
11    Class DAB{
12           .....
13           Bankcenter b = new Bankcenter();
14           public void IdentifyPinCode(receivedCode) {
15           String val = ''BadClient'';
16           try{ b.verifyCode(receivedCode);}
17           catch(ClientinBadlistException e){ this.swallowcard(); return val;}
18            } }
```

The pseudo code seen in the Listing.5 presents the return handling mode shown in the figure.23. This handling mode is expressed in the try/catch bloc included in this code portion.

AbortAll stereotype Because exception handling can cause software execution stop we have defined this stereotype in order to express abort all handling mode. With the notation shown in the figure.24, the designer can directly put the shown square in the bottom of the box representing the handler to say that such handler will make exit to the system.
The figure.25, represents the sequence diagram that describes the interactions of payment between a client and a broker in the travel agency application. We have already used this example to show the usefulness of retry handling mode. But here, we use it to express the abort all handling mode. After three times of IncorrectCode exception occurrence, the handler must abort all because it predicts that the client try to hacker the system.

Figure 24: Graphic notation of AbortAll stereotype

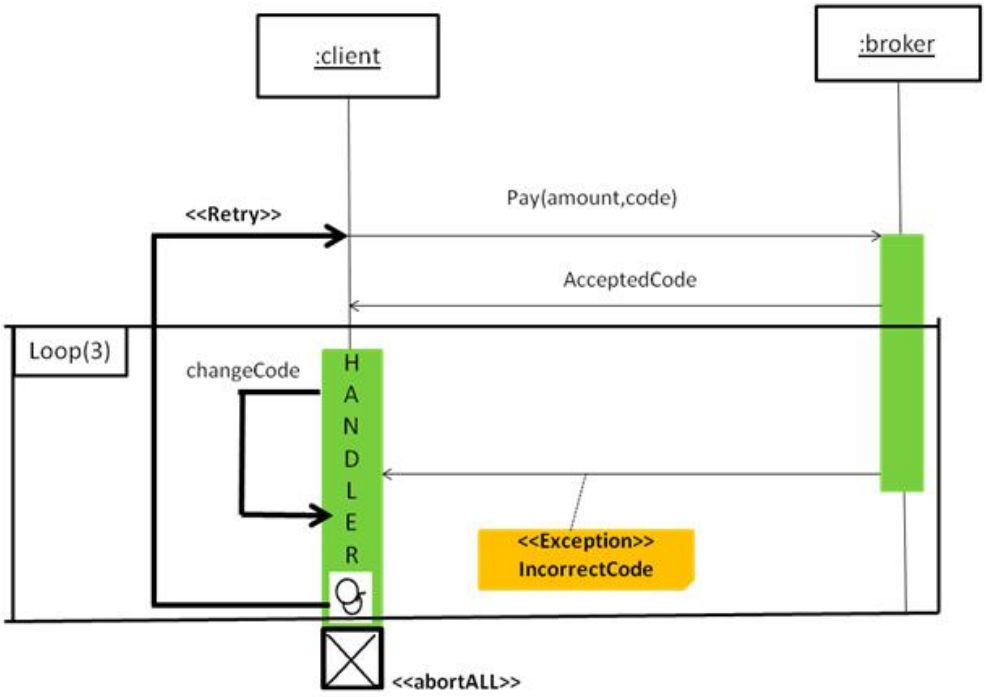

Figure 25: Application of abortAll handling mode

The pseudo code seen in the Listing.6 demonstrates how expressing in the software code the specified AbortALL handling mode made in the sequence diagram.

We think that adding this new specification to UML will be interesting, because designers will not behave with handlers as a piece of code that is specific for a programming language feature any more , but they will behave with handlers as a concrete graph with a high level of flexibility and explicitness .

4.2.5 Step5: defining class diagram

In this step, the designer is demanded to define class diagram in order to describe the static part of the software. In order to facilitate this task, we propose to generate the class diagram from the sequence diagrams. We have not proposed any extension to UML that allows expressing exception handling graphically in the class diagram. But we find that this step is necessary because it represents an intermediary passage from sequence diagram to the software code. In addition, in this step we can associate each class operation

Listing 6: The abortAll handling mode

```
1
2
3
4
5     Class Broker{
6         .....
7         .....
8         .....
9         public void enterCode(int code) throws IncorrectCodeException{....} }
10
11    Class Client{
12        .....
13        int i = 0;
14        Broker b = new Broker();
15        public void pay(int cardcode){
16        boolean accepted = false;
17        while(!accepted){
18        i = i + 1;
19        try{ b.enterCode(cardCode);
20        accepted = true;}
21        catch(IncorrectCodeException e){
22        System.out.println("Invalid code, retry again with the right card code");
23        if (i == 3){ System.exit();} }
24
25        } }
26      }
```

to the exceptions that it could raise.

4.2.6 Step6: Implementing code

In this step, the developer should implement the software code and put exceptions and handlers in the right try/catch blocks. To facilitate this task, we propose to generate the software code skeleton. This code skeleton will contains both exceptions and handlers. There will be try/catch blocks which contains the exceptions and handlers defined in the early phases of the software life cycle. The generated code will conforms to the chosen programming language.

We have explained in this section the proceeding of our approach that allows the designer to express exception handling in the software life cycle. Our approach offers to the designer the opportunity to manipulate exception handling graphically in order to help him verifying the software reliability easily.

5 Implementation of the Proposed approach

In order to implement our approach we were based on model driven engineering to manipulate all the UML models and diagrams which we have used and the new extensions which we have done to UML. To increase the benefits of our approach, we have proposed a tool that facilitates software development by automating its most important phases. This tool looks also after moving Exception Handling defined in the first phase of software life cycle to the next phases until arriving to implementation phase automatically.

We have used the development environment Eclipse Kermeta in order to implement our approach and the Eclipse UML2 plugin to edit and view all the UML diagrams that we needed. The subsection 5.1 introduces the Model driven engineering. Subsection 5.2 presents an overview about kermeta language, subsection 5.3 describes the eclipse UML2 plugin and the last subsection of this section analyses the architecture of the proposed tool in addition to the most important transformations and their implementations with kermeta.

5.1 Model-driven engineering

According to [17] Model-driven engineering (MDE) is a software development methodology which focuses on creating models, or abstractions, more close to some particular domain concepts rather than computing (or algorithmic) concepts. It is meant to increase productivity by maximizing compatibility between systems, simplifying the process of design, and promoting communication between individuals and teams working on the system.
A modeling paradigm for MDE is considered effective if its models make sense from the point of view of the user and can serve as a basis for implementing systems. The models are developed through extensive communication among product managers, designers, and members of the development team. As the models approach completion, they enable the development of software and systems.
As it pertains to software development, model-driven engineering refers to a range of development approaches that are based on the use of software modeling as a primary form of expression. Sometimes models are constructed to a certain level of detail, and then code is written by hand in a separate step. Sometimes complete models are built including executable actions. Code can be generated from the models, ranging from system skeletons to complete, deployable products. With the introduction of the Unified Modeling Language (UML), MDE has become very popular today with a wide body of practitioners and supporting tools. More advanced types of MDE have

expanded to permit industry standards which allow for consistent application and results. The continued evolution of MDE has added an increased focus on architecture and automation.

5.2 Used Tools

Kermeta workbench is a powerful metaprogramming environment based on an object-oriented DSL (Domain Specific Language) optimized for metamodel engineering.

5.2.1 Kermeta

According to [18] Kermeta is a metamodeling language which allows describing both the structure and the behavior of models. It has been designed to be fully compliant with the OMG metamodeling language EMOF (part of the MOF 2.0 specification) and provides an action language for specifying the behavior of models.
Kermeta is intended to be used as the core language of a model oriented platform. It has been designed to be a common basis to implement Metadata languages, action languages, constraint languages or transformation language. Kermeta has many features such as:

- Model oriented : Model elements are the key concepts of the language, manipulate them as easily as Objects in your favorite Object Oriented language.
- Aspect oriented : It allows to weave elements coming from various sources (ecore, kermeta, OCL, ...) in order to build various tools on top of existing (legacy) metamodels. Place your operations and features directly in the metaclasses for clean and innovative designs.
- Strongly typed : It supports generics, lambda expression (build your own "foreach" on any element you wish) and a convenient Model type. Common errors are reported early in the development process.
- Design by contract : this helps to build reliable tools.
- Object Oriented core : If the above features aren't enough for a good design, then simply reuse the popular know how and design pattern available with Object Oriented Programming. This ensures a good scalability

5.2.2 Eclipse UML2 plugin

According to [19] UML2 is an EMF-based implementation of the Unified Modeling Language (UMLTM) 2.x OMG metamodel for the Eclipse platform.
The objectives of the UML2 component are to provide:

- a useable implementation of the UML metamodel to support the development of modeling tools
- a common XMI schema to facilitate interchange of semantic models
- test cases as a means of validating the specification
- validation rules as a means of defining and enforcing levels of compliance
- Object Oriented core : If the above features aren't enough for a good design, then simply reuse the popular know how and design pattern available with Object Oriented Programming. This ensures a good scalability

UML2 Tools is a set of GMF-based editors for viewing and editing UML models; it is focused on (eventual) automatic generation of editors for all UML diagram types.

5.3 The Tool's Architecture

Before describing the proposed tool, you should note that the stereotypes we have defined could be applied directly by the Eclipse UML plugin. This could be done after defining the profile and saving it. Stereotypes are just defined by creating extensions to the needed meta classes, then, we can apply them to UML diagrams. The tool we propose consists of three main tasks(see figure.26). The first one consists on generating suitable sequence diagrams for each use case, the second one consists on generating the class diagram from sequence diagram and the final one consists on code generation. With each task there will be a kind of migration made by the exception handling system from one phase to an other. So, each time there is a generation the exception handling specified in each phase will move automatically to the next phase. In the following we will analyze each task separately in order to get a clear idea about each one and about the transformations made for passing from one diagram to an other. We will also explain in each task the exception handling aspect that characterizes it in order to keep our target context that looks after exception handling in software life cycle.
The implementation of the transformations will be given each time we explain a task.

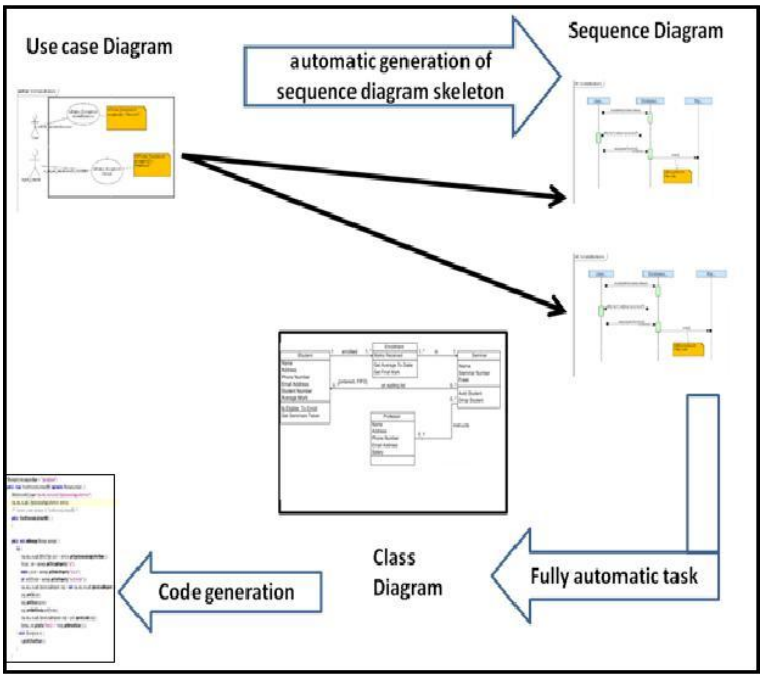

Figure 26: Main tasks of our proposed tool

Task.1 : Generating sequece diagrams skeletons

Input: Use case diagram based on the proposed profile

output : Sequence diagrams skeletons

Description The use case diagram used as an input is to be defined by the designer. He must apply to it the profile described above. That means that he has to apply the defined stereotypes that can be applied to the use case diagram such as ≪WithExceptions≫ and ≪AssociatedExceptions≫ that are to be applied respectively to use cases which interactions can raise exceptions and to comments which annotate those use cases.
Our tool will produce one sequence diagram skeleton for each use case. Then the designer is asked to add messages, interactions and the right lifelines(see figure.27). He can also add other sequence diagrams in order to satisfy the needed requirements. In this task, each exception contained in a comment stereotyped by ≪AssociatedExceptions≫ in the use case diagram will be mapped to a comment stereotyped by ≪Exception≫ in the sequence diagram. This subtask is done automatically. The developer must represent in

the sequence diagram each exception by a reply in the right place of raising context of the exception.
The proceeding of this task as shown by is divided into four subtasks(see figure.27):

1. In this subtask there will be automatic generation of empty sequence diagrams. We will have for each use case a sequence diagram that will take its name.
   Each comment stereotyped by ≪AssociatedExceptions≫ in the use case diagram will be dissociated to be replaced in the sequence diagram by many comments stereotyped by ≪Exception≫ and each one represents the name of an exception.

2. In this subtask, the designer is asked to make the needed refinements such as life lines, messages ...etc. But the most important thing is placing exceptional replies in the right place. Then he must rename each exceptional reply by the right exception name. Finally, he has to define a handler to each exception according to the Handler specifications we have presented in subsection4.4.

3. In this subtask, our tool will mark automatically the exceptional replies by comments stereotyped by ≪Exception≫. This allows the designer distinguishing between normal replies and exceptional replies. It facilitates also the transformation to the class diagram because our tool will create for each exceptional reply in the sequence diagram an exception class and put it in a package which contains all exception classes.

4. In this subtask, the designer has to define the right handler for each exception according to our proposed handler specification.

Implementation To realize this task we have implemented with kermeta an Operation that gets as inputs two UML models. The first one is the source that represents the use case diagram augmented with our proposed extensions. The second one represents the model that will receive the informations representing the sequence diagrams skeletons. The last UML model will receive also the set of exceptions associated to exceptional uses cases . The kermeta operation we are talking about, will transform each use case to a sequence diagram skeleton and it will transform each comment stereotyped with AssociatedExceptions to a set of comments stereotyped by Exception. Each of these comments will receive an exception name and will be included to the right sequence diagram skeleton.

The following code portion represents the kermeta operation that is responsible for the first transformation. We mean here the transformation from use case diagram to sequence diagram.

Listing 7: Transformation.kmt(transformation from a use case diagram to sequence diagrams skeletons)

```
1  operation transformerUscToSeq(SourceModel : Model,TargetModel : Model ) : Void is do
2   var p : Package
3   var c : Collaboration
4   var i : Interaction
5   var comment : Comment init Comment.new
6   comment.body := ""
7   var s : String init ""
8   var Setcomment : set Comment[0..*] init kermeta::standard::Set<Comment>.new
9   getAllUCases2(usc.asType(Package)).each{ uc |
10  p := Package.new p.name := uc.name
11  c := Collaboration.new c.name := p.name+"Collaboration"
12  i := Interaction.new i.name := "sd"+p.name
13  getAllComment(usc.asType(Package)).each{com |
14  com.annotatedElement.each{elt |
15  if elt.asType(UseCase) == uc
16  then from var j : Integer
17  init 0 until j == com.body.size()  loop
18  if com.body.elementAt(j).toString() == ">"
19  then from var k : Integer init j + 2 until k == com.body.size()
20  loop s := s+com.body.elementAt(k).toString()
21  if com.body.elementAt(k+1).toString() == "/"
22  then   comment := Comment.new  comment.body := "<<Exception>>"+s
23  if not i.ownedComment.exists{c | c.body == comment.body }
24  then  i.ownedComment.add(comment) end s :="" k := k + 1 end  k:= k + 1 end
    end
25  j := j + 1     end
26  end   } i.ownedComment.add(com) }  c.ownedBehavior.add(i)
27  p.packagedElement.add(c)  TargetModel.packagedElement.add(p) }
28
29  end
```

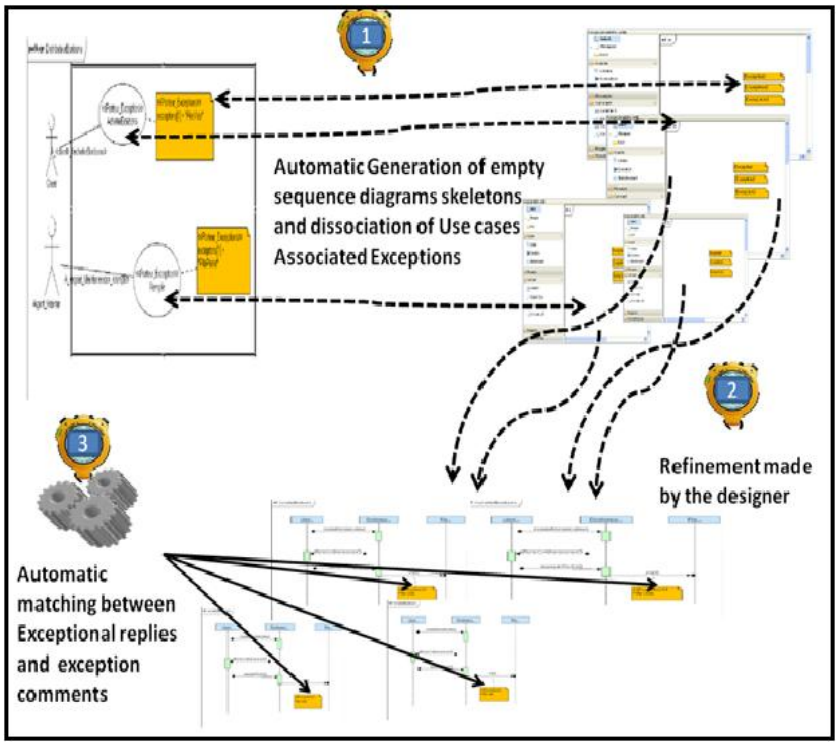

Figure 27: Main subtasks of task.1

Task.2 : Generating Class diagram

Input: Sequence diagrams

output : Class diagram

Description In this task, the tool will generate a class diagram from sequence diagrams defined in the previous task. This task is divided into tow subtasks(see figure.28) :

1. In this subtask there will be automatic generation of a class diagram that describes the static side of the application. There will be also automatic generation of different operations in the right classes and the associations between classes. Each comment stereotyped by ≪Exception≫ will be replaced in the class diagram by a class that takes the name of the exception.
   The classes representing exceptions will be put in a separate package which name is ExceptionalPackage.

2. In this subtask, each operation will be matched automatically with the raised exceptions which it can raises.

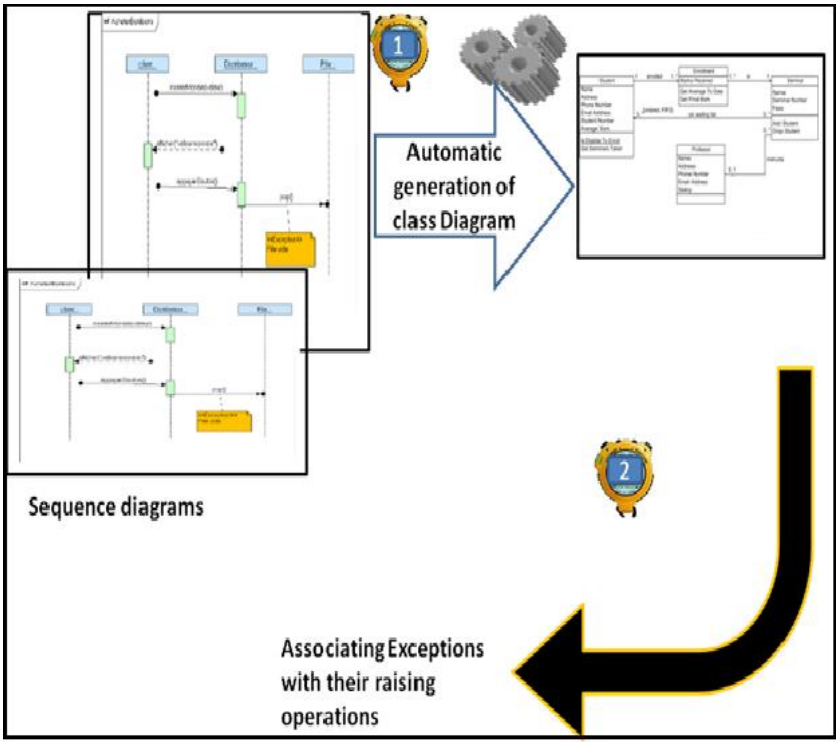

Figure 28: Main subtasks of task.2

Implementation We have implemented the transformation from sequence diagrams to a class diagram with a Kermeta Operation that takes as input a set of sequences diagrams saved as a UML model and a UML model that will save the generated class diagram. Each life line in a sequence diagram will be transformed to a class. The operation will delete the redundant classes, because the set of the sequences diagram may contain life lines with the same name. Each message will be transformed to a method which be placed in the right class (you should note here that the life line that will receive the message will guide the tool to place methods in their right owners). Each exception in the sequence diagram will be transformed to an exception class in the target UML model.

The following code portion represents the kermeta operation that is responsible for the second transformation (The transformation from sequences diagrams to a class diagram).

Listing 8: Transformation.kmt(Class diagram generation)

```
1  operation generateClassDiagram(Msource : Model , Mcible : Model ) : Void is do
2   var c : Class
3   var o : Operation
4   var Excl : Class
5   var pacExceptions : Package init Package.new
6   var InterMessages : seq Message[0..*] init kermeta::standard::Sequence<Message>.new
7   var AnnotedElements : Element[0..*] init kermeta::standard::OrderedSet<Element>.new
8   pacExceptions.name := "MyExceptions"
9
10  getAllPackage(Msource).each{p |
11  getAllLifeLines(p).each{l | c := Class.new c.name := l.name
12  getAllInteractions(p).each{i | i.ownedComment.each{com|
13  com.annotatedElement.each{ ae | AnnotedElements.add(ae) }
14  Excl := Class.new Excl.name := com.body
15  if not pacExceptions.packagedElement.exists{pac |
16  pac.name == Excl.name }  then pacExceptions.packagedElement.add(Excl) end}
17  InterMessages  := i.message.select{m |  m.receiveEvent != void }
18  InterMessages.each{ im |
19  o := Operation.new o.name := im.name
20  im.receiveEvent.asType(MessageOccurrenceSpecification).covered.each{ lf |
21  if (lf  == l) and (not AnnotedElements.exists{aelt |
22  aelt.asType(Message).name == o.name })
23  then c.ownedOperation.add(o)   end } }}
24  if not getAllClasses(Mcible).exists{classe | classe.name == c.name}
25  then Mcible.packagedElement.add(c) end  } }
26  Mcible.packagedElement.add(pacExceptions)
27  saveModel(Mcible , "platform:/resource/Stage2/model/SAGECLcible.uml")
28  genererAssoc(Msource , Mcible)
29  saveModel(Mcible , "platform:/resource/Stage2/model/SAGECLcible.uml")
30
31 end
```

As we did not make any additional extension to UML to describe exception handling graphically in the class diagram, we were based just on what UML proposes. The exceptions will be automatically associated to the raising contexts. So, each exception will be associated to the operation that could raise this exception.
We are planning to look for future extensions to UML to give designers the possibility of designing exception handling explicitly in the class diagram.

Task.3 : Generating Code

Input: Class diagram

output : JAVA Code

Description This task represents the final task and it is considered the most important one because with it we could evaluate the efficiency of our tool. It is considered also as the productive task. The developer will receive after the running of this task the code of the application that has been specified in the early phases of software life cycle. The output of this task is a code that contains all classes and operations skeletons with the whole exception handling system specified previously put in the right places .
The code generated will be produced according to the chosen programming language. In this task, we will see explicitly the exceptions and their handlers put in the right places. Exception handling will respect the target programming language features and its syntaxe. Syntacticly, exception handling system will be automatically generated according to the target programming language feature such as try..catch in JAVA , with : do : in SmallTalk....etc. In addition exception handling system specified by the designer in the early phases will be tested in this task by our tool according to the target programming language features. For the moment we can only generate JAVA code, but we plan to aim other programming languages.

Implementation To generate code we have defined a Kermeta Operation that takes as an input a class diagram and then generate the code skeleton. This code skeleton will contains class definition, methods bodies the exceptions associated to their raised methods and the try/catch blocs. In addition, the code skeleton will contains the definition of exceptions classes which are inherited from the class Exception of JAVA. The handlers are to be made by the developers here, because we have not implemented the handler specifications seen in the subsection 4.5 yet. But this is considered as a work in progress which could be continued as a future work.

The following code portion represents the kermeta operation that generates the code skeleton with the specified exceptions from a class diagram.

Listing 9: Transformation.kmt(Code Generation)

```
operation generateCode(classDiagram : Model , TargetLanguage : String ) : Void is do
 var h : String
 var opr :String
 var ch  : Character
 var aff : String
 var s  : String init ""
 var m : String
 var t : String
 var c : String

 if TargetLanguage == "Java" then t := "try" and c := "catch" end
 classDiagram.each.{c | c.ownedOperation.each{op |
 if op.raisedException.size() != 0
 then from var i : Integer
 init 0 until i == op.raisedException.one.name.size()
 loop s:=s+op.raisedException.one.name.elementAt(i).toString()
 if s =="<<Exception>>"
 then s := "" from var j : Integer
 init i+1 until j == op.raisedException.one.name.size()
 loop  s := s+op.raisedException.one.name.elementAt(j).toString() j := j +1 end
 op.raisedException.one.name := s end  i := i +1 end  end }
 stdio.writeln("//----------Class skeleton of "+c1.name)
 stdio.writeln("")
 stdio.writeln("")
 stdio.writeln( "public Class   "+c1.name+" {") stdio.writeln("")
 stdio.writeln("")
 stdio.writeln("")
 c.ownedAttribute.each{att | stdio.writeln(att.name+"  "+att.name+" ;")
 stdio.writeln("") stdio.writeln("")}
 stdio.write("public  "+c1.name+"(") c1.ownedAttribute.each{att |
 stdio.write(att.name+"  "+att.name+" , ")}
 stdio.writeln(" ){") c1.ownedAttribute.each{att |
 stdio.writeln("this."+att.name+" := "+att.name+";" ) }
 stdio.writeln("// TO DO //  }")
 stdio.writeln("")
 stdio.writeln("")
 c.ownedOperation.each{op |
 if op.raisedException.size() != 0
 then stdio.writeln( "public void  "+op.name+" () {" )


          stdio.writeln("{"+t+"  { // To DO //}")
          stdio.writeln("//      !      ")
          stdio.writeln("//   !   !   ")
          stdio.writeln("//  !      !")
          stdio.writeln("// !       !")
          stdio.writeln("//! Exception!")
          stdio.writeln("//! Localisation!")
          stdio.writeln(" //!!!!!!!!!!!!!!!!")
```

```
50            stdio.writeln("")
51            stdio.writeln("")
52            stdio.writeln(c+"("+op.raisedException.one.name+" e){// TO DO //}}")
53            stdio.writeln("")
54            stdio.writeln("")
55            stdio.writeln("")
56
57   else stdio.writeln("public void   "+op.name+"(){// TO DO //}   }")
58            stdio.writeln("")
59            stdio.writeln("") end    }}
60            stdio.writeln("")
61            stdio.writeln("")
62
63  end
```

In this section we presented The tasks done by our proposed tool. Even we have implemented most of the transformations cited above, the implementation of handlers specifications is in progress.

6 Conclusion and perspectives

We have performed in this work some extensions to UML in order to give opportunity to designers to deal with exception handling in the early phases of software life cycle. We have done our proposed extensions according to a defined profile that extends UML in order to introduce exception handling concepts, notations and terminology in UML diagrams. This feature does not exist in standard UML. Extensions we have done have been applied to use case and sequence diagrams. These extensions are translated to graphical notations that make sense for designers. Designers have looked always after making standard use case diagram and defining standard interactions or sequence diagram, then the developer will look after exception handling in the last phase of software life cycle. Even developer could sometimes extract all exceptions and define their handlers efficiently, but this is dependent on the kind of the software that he implements. In large size applications where developer is found under pressure because of a huge number of code lines, he could forget discovering some exceptions or defining some handlers. Our approach push the designer to avoid these mistakes, because the difficulty of making a reliable software will be divided between designers and developers and dissociated along the software life cycle.This gives possibilities of software reliability verification in each phase.
With the extensions we have done, use case and sequence diagram will be enriched by exception handling notations, a thing that provides explicitness of exception handling which was always considered as a strength piece of code. With our proposed UML extensions, designers and developers can manipulate and deal with exception handling from the early phases of software life cycle and with some level of explicitness and graphic notations. In addition to our proposed extensions we have implemented our approach with a tool which help both designers and developers not only to deal with exception handling in all software life cycle but also to help at generating sequence diagrams, class diagrams and code.
As perspectives we are planning first to go ahead with our proposed extensions in order to deal with exception handling notation in the class diagram. As we have mentioned above, we have dealt with exception handling in class diagram by benefiting from the feature of putting exceptions in the right raising contexts offered by UML. But, we will focus on extending UML to support graphic notations of exception handling in class diagrams. This will allows designers to manipulate graphically exceptions, their raising contexts and their handlers in the class diagram. We look also for completing the implementation of handlers specification made for sequence diagrams. As a future work, we plan to look for dealing with exception handling in components.

References

[1] Goodenough, J.B.: *Exception handling: Issues and a proposed notation.* Communications of the ACM 18(12), 683–696 (1975).

[2] Shui, A., Mustafiz, S., Kienzle, J.: *Exceptional use cases.*In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 568–583. Springer, Heidelberg (2005).

[3] Shui, A., Mustafiz, S., Kienzle, J.: *Exception-Aware Requirements Elicitation with Use Cases.* In: Dony, C., Knudsen, J.L., Romanovsky, A., Tripathi, A.R. (eds.) Advanced Topics in Exception Handling Techniques. LNCS, vol. 4119, pp. 221–242. Springer, Heidelberg (2006).

[4] Mustafiz, S., Sun, X., Kienzle, J., Vangheluwe, H.: *Model-Driven Assessment of Use Cases for Dependable Systems.* In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 558–573. Springer, Heidelberg (2006)

[5] Zia, M., Mustafiz, S., Vangheluwe, H., Kienzle, J.: *A Modelling and Simulation Based Process for Dependable Systems Design.* In: Software and Systems Modeling, pp. 437–451 (April 2007).

[6] Romanovsky, A.: *On Exceptions, Exception Handling, Requirements and Software Lifecycle.* 2007 IEEE.

[7] Si Alhir, S: *Guide to applying the UML,* page 350. Springer, 2002 / Si Alhir, Sinan (2002). Guide to applying the UML. Springer.

[8] Oddleif, Halvorsen., Ragnhild Kobro, Runde., Oystein, Haugen.: *Time Exceptions in Sequence Diagrams* MoDELS 2006 Workshops, LNCS 4364, pp. 131–142, 2007. Springer.

[9] Nelio, Cacho., Thomas, Cottenier., Alessandro, Garcia.:*Improving Robustness of Evolving Exceptional Behaviour in Executable Models* WEH '08, November 14, Atlanta, Georgia, USA, 2008. ACM.

[10] OMG. *Catalog of OMG Modeling and Metadata Specifications* Retrieved 2008-03-31.

[11] Rohrig R, Beutefuhr H, Hartmann B, Niczko E, Quinzio B, Junger A, Hempelmann G.: *Summative software evaluation of a therapeutic guideline assistance system for empiric antimicrobial therapy in ICU.* J Clin Monit Comput 2007; 21:203–210.

[12] Ian, Alexander. *Use Cases with Hostile Intent* IEEE Software, January 2003.

[13] de Lemos, R., Romanovsky, A.: *Exception Handling in the Software Lifecycle.* International Journal of Computer Systems Science and Engineering, 16 (2). pp. 167-181. ISSN 0267-6192.

[14] Dowson, M.: *The Ariane 5 Software Failure.* Software Engineering Notes 22 (2): 84. doi:10.1145/251880.251992.

[15] C. M. F. Rubira, R. de Lemos, G. R. M. Ferreira, F.Castor Filho. *Exception handling in the development of dependable component-based systems.*Software – Practice and Experience. 35. 2005.

[16] Miller, R.: *What's New in UML2? Model Exceptions.* Embarcadero Network. http://edn.embarcadero.com/article/30166.

[17] Schmidt, D.C.:*Model-Driven Engineering* IEEE Computer, 2006.

[18] Franck, C., Zoe, D., Franck, F. (2007). *Kermeta Language Overview*

[19] Eclipse Foundation (2010). *Model Development Tools* http://www.eclipse.org/modeling/mdt/?project=uml2.

[20] Christophe, Dony.: *A fully object-oriented exception handling system: rationale and smalltalk implementation* Springer-Verlag New York, Inc. New York, NY, USA, 2001.