# EXCEPTIONAL USE CASES

AARON FU-SHEN SHUI

SCHOOL OF COMPUTER SCIENCE

MCGILL UNVERSITY

MONTREAL, QUEBEC, CANADA

SEPTEMBER 28, 2005

A THESIS SUBMITTED TO MCGILL UNIVERSITY IN PARTIAL

FULLFILMENT OF THE REQUIREMENTS OF THE DEGREE OF

MASTER OF SCIENCE

*Your file Votre référence*
*ISBN: 978-0-494-22765-7*
*Our file Notre référence*
*ISBN: 978-0-494-22765-7*

# ACKNOWLEDGMENTS

There are many people I owe a debt of gratitude to in relation to this thesis. First and for most, I would like to thank Professor Jörg Kienzle who's outstanding supervision and brilliant direction inspired me to work toward achieving highest standards of academic excellence. Professor Christophe Dony at the Université de Montpellier contributed his invaluable knowledge and experience on exceptions to my work. I thank my peers, Sadaf Mustafiz and Alexandre Denault. Sadaf has provided a great deal of advice as an expert on fault tolerance, and Alex translated the thesis' abstract into French. Among my family and personal friends, I thank my mother, Ping-Chii Shih and my uncle Yaohuaui Shen. Their guidance and support through my life have allowed me to pursue my ambitions and made all my achievements possible. Finally, I must thank Victoria Yang for her unconditional encouragement and support; I could not have written this thesis without her.

# TABLE OF CONTENTS

# LIST OF FIGURES

# ABSTRACT

Many exceptional situations arise during the execution of an application. When developing dependable software, the first step is to foresee these exceptional situations and document how the system should deal with them. This thesis outlines an approach that extends use case based requirements elicitation with ideas from the exception handling world. After defining the actors and the goals they pursue when interacting with the system, our approach leads a developer to systematically investigate all exceptional situations arising in the environment or in the system that change or fail user goals. Means are defined for detecting the occurrence of all exceptional situations, and the exceptional interaction between the actors and the system necessary to recover from such situations is described in handler use cases. To conclude the requirements phase, an extended UML use case diagram summarizes the standard use cases, exceptions, handlers and their relationships.

# ABRÉGÉ

Plusieurs situations exceptionnelles peuvent se produire pendant l'exécution d'une application. Pour développer un logiciel sûr et robuste, il faut premièrement prévoir ces situations exceptionnelles et documenter comment le système devrait leurs réagir. Ce document décrit une technique qui améliore les analyses de besoins avec des cas d'usage en y ajoutant des idées de gestion d'exception. Après avoir défini les acteurs et les buts d'interaction avec le système, le programmeur doit enquêter toutes les situations exceptionnelles qui pourraient se produire dans le système (les situations exceptionnelles se produisant dans l'environnement qui peuvent changer les buts de l'utilisateur et les situations exceptionnelles reliées au système qui menacent les buts de l'utilisateur). Des procédés sont proposés pour reconnaître la présence de situations exceptionnelles et les interactions acteurs/système qui rétablissent le système après une des situations soit illustrées dans les cas d'usage "gestionnaires". Pour conclure la phase d'analyse de besoins, un diagramme étendu UML des cas d'usage résume les cas d'usage standard, les exceptions, les gestionnaires et leurs relations.

# CHAPTER 1: INTRODUCTION

Most main stream software development methods define a series of software development phases: requirements elicitation, analysis, design and finally implementation, which lead the development team to discover, specify, design and implement the main functionality of a system. While the system's main functionality dictates the system's behaviour most of the time, special situations may arise during the execution that call for processing beyond the main functionality. Failure to recognize and specify how the system responds in such situations jeopardizes how well the system can perform its main functionality. Unfortunately, when using a standard software development process, there is no guarantee these situations are considered during development. How well the system handles these situations depends highly on the imagination and experience of its developers. As a result, the implementation might not function correctly under many probable situations.

When developing dependable systems, e.g. safety-critical systems, where a malfunction can cause significant or unacceptable damage, nothing should be left to chance. Following the idea of integrating

exception handling into the software development life cycle [Goodenough 1975, De Lemos 2001], this thesis describes an *exception-aware* approach to *use case based* requirements elicitation that leads developers to consider the adverse or *exceptional* situations a system under development might be subjected to. It is very important to think about exceptional behaviour at the requirements phase, because it is up to the users and stakeholders of the system to decide how they expect the system to react to exceptional situations. Only with exhaustive and detailed user and stakeholder feedback is it possible to discover and then specify the complete system behaviour in a subsequent analysis phase, and decide the need for employing fault masking and fault tolerance techniques for achieving runtime reliability during design.

Chapter 2 provides background information of use cases and exception as they traditionally appear. In Chapter 3, the ideas from the exception handling world are re-interpreted in the context of use cases. These redefined use case exceptions are accommodated through various extensions to use case diagrams and descriptions, which are described in Chapter 4. These extended use case diagrams and descriptions are used to document the exceptional behaviour captured through the exception-

aware process, which is discussed in Chapter 5. Chapter 6 presents a case study of an Elevator Control System to illustrate how the exception-aware process is applied to yield exceptional use cases. Related works and future work are discussed in Chapter 7 and 8, respectively, and the last chapter draws some conclusions.

# CHAPTER 2: BACKGROUND

This section gives a brief overview of conventional exceptions and standard UML use cases. In addition, the key elements of an exception handling system are discussed, and version of a textual use case template is introduced.

## 2.1. Exceptions

Traditionally, exceptions have been a feature of programming languages and as such, usually considered only during the later phases of software development, i.e. low-level design and implementation. Exceptions are supported and realized in various ways across different programming languages and/or systems. Nevertheless, the fundamental idea and goal remains the same; the concerns of error detection and correction are separated from the primary functionality of a program, and initiated when necessary by exceptions.

Generally defined, an *exception* is an indicator that is raised when a *signaller* detects an error. An *exception occurrence*, subsequently,

requires a computation outside the current scope or context of the program for the program to continue [Knudsen 1987]. The normal flow of a program is *interrupted*, and control is passed to an *exception handler*, which executes *exceptional behaviour*. Alternatively, a handler can *ignore* the exception, or just *propagate* (i.e. pass the exception on to the enclosing context) the exception as is, or even signal a new exception. After the handler has executed, the system returns control to the original context (*resumption mode*), or terminates the original context (*termination mode*) [Goodenough 1975].

A programming language or system with support for exception handling is called an *exception handling system (EHS)* [Dony 1990]. An EHS provides a way to define and coordinate signallers, exceptions, and handlers. Coordination includes managing the role of signallers, the scope of exceptions, and the activation of handlers. The *context* in which an exception is raised helps determine how it is handled. Depending on the EHS, a context can be: a program, a process, a statement, an expression, etc. EHS's can provide more (or less) sophisticated exception handling support than what is described above according to what is required by their application domain.

## 2.2. UML and Use Cases

The Unified Modeling Language (UML) [Rumbaugh 1999] defines a notation for specifying and documenting the artefacts of a software-intensive system. UML is intentionally process independent. However, it offers various diagrams that unify the scores of graphical modeling notations that existed in the software industry during the 80's and 90's. Among the UML models, this thesis focuses on the *use cases* as defined in the UML 2.0 specification [OMG 2004].

Since their introduction in the late 80's [Jacobson 1987], use cases have become a widely used formalism for discovering and documenting the behavioural requirements of software systems [Larman 2002]. A use case diagram serves to describe a system's responsibilities and interactions with respect to its environment without revealing details of the system's internal workings. Each use case represents a series of interactions, which satisfies a goal of a particular stakeholder or *actor* when successfully completed. Actors are external entities that interact with the system. There are two types of actors, primary actors, which

have goals with the system and secondary actors, which do not. To promote reuse and modularity, use cases can *Include* interactions of another use case, or optionally *extend* another use case. This is represented by a *directed relationship* between two use cases [OMG 2004]. Use case diagrams provide a concise high-level view of some or all use cases in a system. It allows developers to graphically depict what the system must do to fulfill the needs of its actors.

A use case diagram for an ATM machine is shown in Fig. 2.2.1. The primary actor is the *Bank Customer* who uses the ATM machine to *Withdraw Money* or *View Account Status*. The secondary actor, *Bank Repository*, stores the information that is read and updated by the ATM. Both Withdraw Money and View Account Status use cases require the Bank Customer to first authenticate by entering their Bank Card and a PIN, which is represent by the use case *Authenticate*.

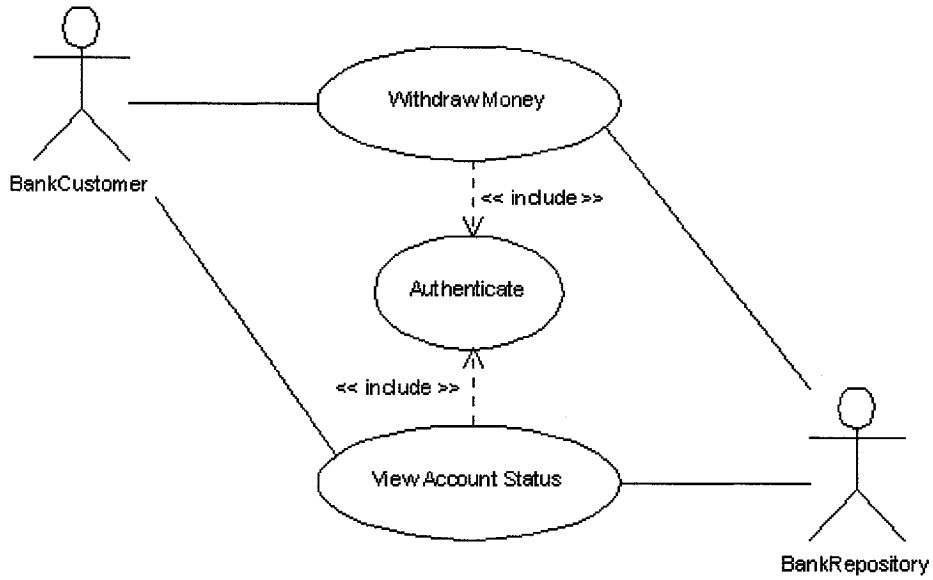


Fig. 2.2.1: ATM Machine Use Case Diagram

Use cases can be described at different levels of granularity [Cockburn 2000]. They can scale up and down in terms of sophistication and formality depending on the needs of developers. At the highest level, *summary* level use cases give an overview of how the system is used. User-goal level use cases describe how the system is used to achieve a user's goal. Finally, sub-function level use cases describe how sub-goals of higher level goals are achieved. Use cases are very effective means of communication between technical as well as non-technical stakeholders of the software under development. Their versatility, coupled with their

ability to document requirements in terms of actor goals, make use cases ideal for requirements elicitation.

The interaction details contained in each use case are not included in the diagram. Some development methods, such as Fondue [Sendall 1999], define a textual template called a *use case description* that developers fill out for each use case. A use case description for the use case Withdraw Money is shown in Fig. 2.2.2. Using the Fondue template forces developers to document all the important features of a use case including: *primary actor*, *main success scenario*, and *extensions*. The main success scenario describes the standard way of achieving the primary actor's goal, while the extensions describe alternate or optional interactions, including ones that lead to failure of the use case by not achieving the goal. The textual use case descriptions work with the use case diagram to provide complete explanation of how the system is expected to work from the actors' point of view.

**Title:** Withdraw Money
**Primary Actor:** Bank Customer
**Intention:** Bank Customer wants to withdraw cash from his/her account.
**Level:** User Goal
**Main Success Scenario:**

1. Bank Customer inserts his/her bank card.
2. System ***Authenticates*** Bank Customer.
3. Bank customer selects the account he/she wishes to withdraw from and enters the amount he/she wishes to withdraw.
4. System sends request to *Bank Repository*.
5. *Bank Repository* processes and approves the request and updates the account balance.
6. System dispenses the requested cash and returns the bank card.

**Extensions:**

2a. Card is invalid. System returns the card. Use case ends in failure.
5a. Bank Repository rejects the request. System notifies Bank Customer and returns the bank card. Use case ends in failure.

Fig. 2.2.2: Withdraw Money Use Case Description

# CHAPTER 3: USE CASE EXCEPTIONS

This chapter offers an interpretation of the exception handling paradigm as it is applied to use cases. Exception handling terminology as defined in this chapter will be referred throughout the rest of this thesis.

## 3.1 Exception Signalling

Exceptions at the use case level are not messages that indicate error, but represent situations that prevent current behaviour of the system from continuing. In terms of use cases, it is a situation that may cause the main success scenario of a use case to fail. When these *exceptional situations* are encountered, the normal behaviour must be interrupted by *exceptional behaviour*, which returns the system to a coherent state.

Exceptional situations arise due to changes in the system's environment or due to errors in the system itself. However, it is rare to discover exceptional situation resulting from system errors at the use case level. Usually, very little is known about the system's internals so it is

difficult to anticipate problems with it. Therefore, most exceptional situations found at the use case level arise from external entities.

Exceptions are used to represent exceptional situations from the system's point of view, whereas exceptional situations take on the perspective of external entities and developers. For example, "max operating temperature exceeded" is the exception signalled for the exceptional situation "cooling system breaks down". It could also be said, the exceptional situation describes the cause, while the exception describes the effect. Often, it is more natural to think about exceptional situations rather than exceptions at the use case level, since use cases are concerned with how actors view and interact with the system. Thinking about exceptional situations leads to the discovery of exceptions.

An exception is signalled when the system encounters an exceptional situation. The signaller can be an actor (*actor-signalled exception*) or a set of conditions the system must somehow check (*system-detected exception*). Actor-signalled exceptions rely on actors to detect exceptional situations, so the system only needs to provide a proper interface or protocol to interpret the signalling actor's intention.

Without knowledge of the system's internals, it is difficult to specify how system-detected exceptions are detected, so a signaller is not necessarily defined. However, it is assumed the system successfully detects and signals the exception. Developers may eventually decide on using specialized (secondary) actors to perform detection.

### 3.2 Exception Handling

Exceptions arise at anytime, and can affect all the currently active use cases in a system. When an exception is signalled, its *context* is the current set of active use cases. An exception activates the necessary handlers for every active use case that is jeopardized by the exceptional situation, which the exception represents. Depending on the exception, some or all of the active use cases are interrupted by exceptional behaviour. The exceptional behaviour is described by one or more *handler use cases (handlers)*. So, the exceptional behaviour must include interaction between actors and the system, or else no handler can be defined. If no handler use case is defined, the use case is not interrupted and thus, will most likely fail.

In the event a handler fails, the interrupted use case will fail by default. Ideally, only one handler should be associated with a specific exception per use case. If more than one is defined, exception handling becomes non-deterministic.

The following summarizes what happens when an exceptional situation is encountered.

1. The exceptional situation is identified and one or more exceptions are signalled.
2. Each exception activates a set of handler use cases which interrupt all use cases sensitive to the exception.
3. After the handler finishes:
   a. the interrupted use cases resume

   OR

   b. the interrupted use cases terminate and fails.

# CHAPTER 4: EXCEPTIONAL USE CASES

This chapter proposes extensions to UML use case diagrams and Fondue use case descriptions to accommodate use case exceptions as described in Chapter 3.

## 4.1 Handler Use Cases

Handler use cases, or *handlers*, are specialized use cases designed to perform exception handling. To distinguish handlers from regular use cases, they are stereotyped *handler*, as shown in Fig. 4.1.1. To further enforce the distinction, handlers may only *include* other handlers and do not *extend* non-handler use case. Similarly, non-handler use cases should not include nor extend handler use cases. Consequently, the normal and exceptional behaviour of a system is cleanly partitioned in a use case diagram; developers can extract and view only the main functionality by hiding all handler stereotyped use cases and their relationships.


Fig. 4.1.1: A handler use case named H1

Handlers activated by actor-signalled exceptions are user-goal driven. An actor signals an exception to ensure their goals with the system do not fail, which includes keeping themselves safe and the system operational. Other handlers play supportive roles to user-goals, i.e. their goal is to prevent a primary user-goal from failing. For example, alerting users and stopping the operation of an over capacity elevator helps avoid hardware failure that prevents users taking the elevator from achieving their goals.

Every handler's use case description includes a new field called *Context & Exceptions*, which describes when the handler is used. Context & Exceptions lists every use cases the handler interrupts, along with the exception that activates the handler. Fig. 4.1.2 demonstrates how Context & Exceptions is used to show that handler H1 interrupts use cases U1 if exception E1 or E2 is raised, and interrupts U2 if exception E1 or E3 is raised.

**Use Case:** H1 <<handler>>
**Context & Exceptions:** U1 {E1}, {E2}; U2 {E1}, {E3}
**Primary Actor:** A1
**Intention:** A1 wants to....

Fig. 4.1.2: Partial use case description for handler H1

## 4.2 Interrupt Relationships

Interrupt relationships are exclusively used to show which handlers interrupt which use cases in a use case diagram. Interrupt relationships are the only way handlers and non-handlers can be associated, because include and extend relationships are forbidden between a handler and a non-handler use case. Interrupt relationships are stereotyped directed relationships similar to include and extend. An interrupt relationship is represented by a dotted arrow with an open arrowhead that extends from a handler case to a use case the handler interrupts. Fig. 4.2.1 demonstrates how an interrupt relationship is used to show handler H1 interrupts the use case U1 when an exception occurs.

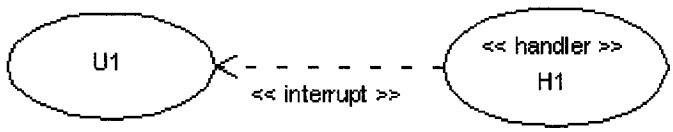


Fig. 4.2.1: Interrupt relationship example

There are two subtypes of the interrupt relationships, namely *interrupt & continue* and *interrupt & fail*, to express in the use case diagram what happens to the interrupted use case after the handler finishes. Interrupt & continue indicates the interrupted use case will resume after the handler is finished, while interrupt & fail indicates the interrupted use case will terminate and fail.

Interrupt & continue and interrupt & fail are also expressed as stereotypes of a directed relationship. Fig. 4.2.2 gives an example of how they appear in a use case diagram.

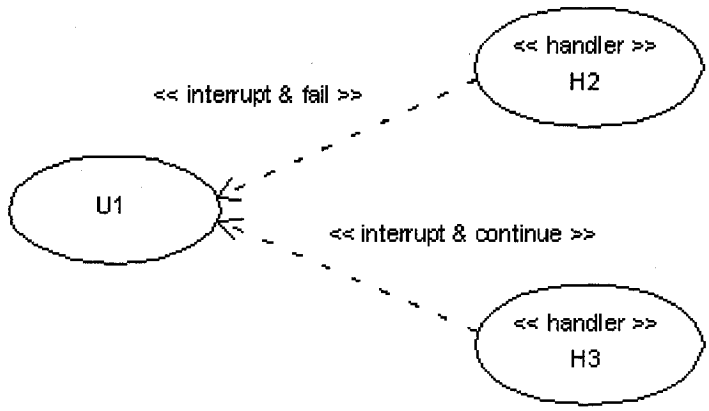


Fig. 4.2.2: Interrupt & continue and interrupt & fail in a use case diagram

The use case diagram in Fig. 4.2.1 only shows one interrupt relationship between a pair of use cases, so when more than one exception is handled and both the resumption and termination modes are used, the relationship carries the more general stereotype, *interrupt*.

To keep the use case descriptions consistent with the use case diagram, there must be a corresponding entry in Context & Exceptions for every interrupt relationship in the use case diagram and vice versa. It should also be stated for every entry whether the use case is interrupted & continued or interrupted & terminated for each exception as shown in Fig.4.2.3.

**Use Case:** H1 <<handler>>
**Context & Exceptions:** U1 {E1 <<interrupt & continue>>}, {E2 <<interrupt & fail>>}; U2 {E1 <<interrupt & fail>>}, {E3 <<interrupt & continue>>}
**Primary Actor:** A1
**Intention:** A1 wants to....

Fig. 4.2.3: Resumption and termination modes added to H1 use case description

## 4.3 Exceptions

In the use case diagrams, the exception(s) that trigger an interrupt relationship are listed on a note associated to the relationship. The exceptions are listed in the note under the heading "Exceptions:". If the relationship is only stereotyped as *interrupt*, then each exception is listed with whether the use case *continues* or *fails*. For example, in Fig. 4.3.1 shows that exception E1 will cause handler H1 to interrupt and continue use case U1 while E2 will cause H1 to interrupt and fail U1.

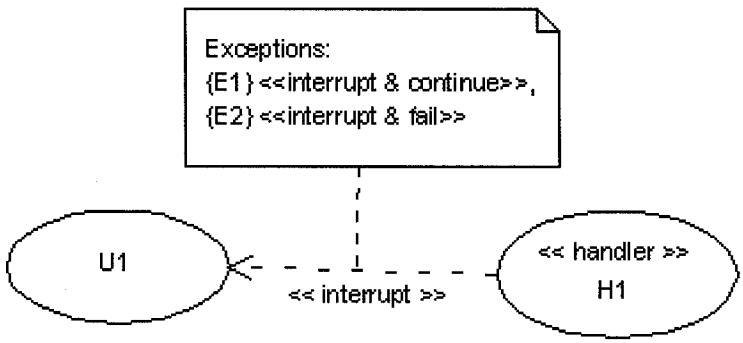


Fig. 4.3.1: Exceptions in an associated note

Some extensions of a use case can cause an exceptional situation and result in an exception. In this case, the resulting exception is listed in the extensions section of a use case description using the syntax *Exception {ExceptionName}*. When a use case's extension causes an exception, it does not mean that use case will be interrupted, because the exception may trigger unrelated handlers. Conversely, not every exception that interrupts the use case is linked to an extension; they may be triggered in an unrelated context. Finally, exceptions should never be triggered by interactions in the main success scenario because then there would be something fundamentally wrong with the use case.

It is easy to confuse exceptions and extension because they are both used to describe alternative behaviour to the main success scenario.

While there are common situations in which exceptions and extensions are used, they are very different concepts. Exceptions serve the purpose of announcing that something is wrong, whereas extensions describe alternative interactions (possibly because something is wrong). Thus, the function of an extension is more comparable to a handler which also specifies alternative interactions. However, extensions describe all alternative interaction steps, but not all alternative interactions leads to use case failure. Handlers are used only when interactions will lead to failure. Furthermore, exceptions do not only announce when alternative interactions jeopardize the success of a use case, they can be signalled for reasons outside the scope of any particular use case.

### 4.4 Exception Table

The exception table is used to list information about all exceptions in the system. So far, information about exceptions have been scattered throughout the use case diagram, and in the extensions and context & exceptions sections of interrupted and handler use case descriptions, respectively. The purpose of the exception table is to consolidate all this information for each exception. The exception table also provides a place

to describe an exception's corresponding exceptional situation, and state whether the exception is actor-signalled or system-detected. If an exception is system-detected, the exception table is a good place to jot down suggestions for detection. Other columns in the table include the use cases interrupted by the exception and the handlers activated by the exception. In addition, exceptional situations discovered that have no defined handlers due to the absence of actor-system interactions can be documented in the table. Finally, besides organizing information about exceptions, for requirement elicitation, the exception table can be extended in a later phase to observe and ensure the exception elements are mapped correctly. Fig. 4.4.1 shows a sample exception table.

| Name | Exceptional Situation | Context | Handler | Detection | Comments |
|---|---|---|---|---|---|
| E1 | Actor does action A. | U1, U2 | H1 | System-Detected<br>Suggestions: check x of component y | Happens often, consider making normal feature |
| E2 | Events C and D occurred. | U2 | H2 | Actor-Signalled | Unlikely but critical. |
| N/A | Component B Overheats | | | System-Detected<br>Suggestions: thermostat? | Not practical to handle, cooling too expensive |

Fig. 4.4.1: Sample exception table

## 4.5 Extending the UML Use Case Metamodel

This section presents an extended UML 2.0 Use Case Diagram Metamodel (Fig. 4.5.1) in the Class Diagram formalism that supports the new exception constructs described in this chapter.

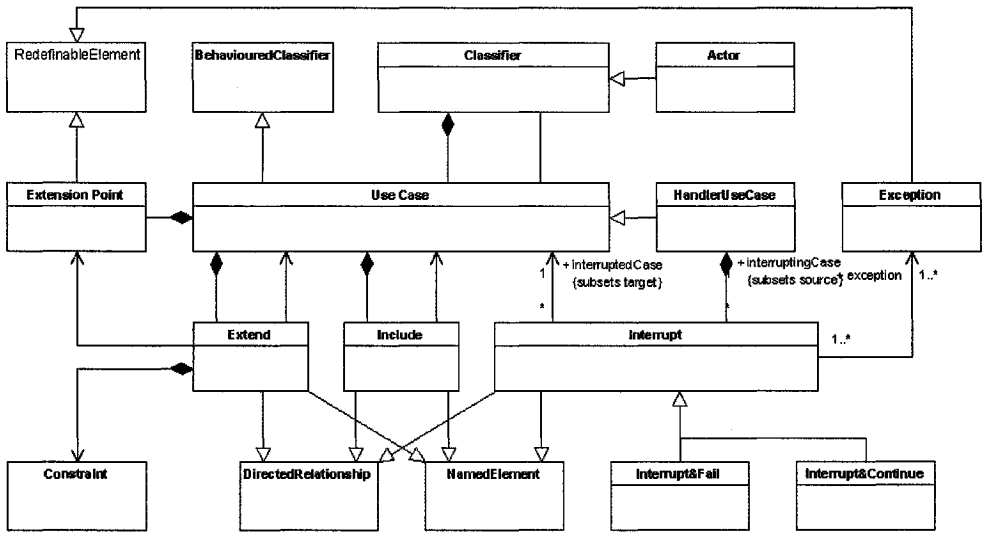


Fig. 4.5.1: Extended Use Case Diagram Metamodel (UML 2.0)

As shown in Fig. 4.5.1, Handler Use Cases are a subclass of Use Cases, while Interrupt relationships behave similarly to Extend and Include relationships except the source must be a Handler Use Case. Finally, Exceptions inherit from Redefinable Element following the same manner as Extension Points.

## 4.6 Failures and Exceptions Revisited

This section reviews and clarifies what happens when an exception is raised and how subsequent failures may manifest themselves and propagate through the system.

As previously defined, exceptions are raised when anticipated problems are detected in the system, and it is always assumed the designated handler use case will successfully correct or recover from the problem. During the execution of a handler, additional exceptional situation may arise and be detected resulting in additional exceptions. However, a use case that is currently interrupted by one handler use case can not be interrupted by another one. Two handler use cases can not concurrently interrupt the same instance of a use case. Additional handlers must either interrupt, extend or be included by the currently active handler use case.

Following the execution of a handler, the interrupted use case may resume (Interrupt & Continue) or fail (Interrupt & Fail). In the case of Interrupt & Fail, the interrupted use case fails, typically, other use cases

including or extended by it will fail as well. Similarly, higher level use cases (e.g. summary-level use cases) that contain the interrupted and failed use case will fail as well. Unless a higher level goal is unusually structured so it does not depend on the success of its sub-goals or there are exceptions and handlers to catch and prevent the propagation of failure. If there are no safeguards, the failure will propagate to the highest level as it would with regular non-exceptional use cases, eventually causing the user goal or even the system goal to fail.

# CHAPTER 5: EXCEPTION-AWARE PROCESS

This section presents an exception-aware approach to use case based requirements elicitation that employs the extended use case diagrams and descriptions and exception table presented in Chapter 4.

## 5.1 Describing Normal Behaviour

The process begins by defining and documenting the primary functionality of the system with use case diagrams and descriptions. The actors and the goals they pursue when using the system, under normal circumstances, are defined and the interactions involved in achieving these goals are captured in use cases. Alternative interaction steps to achieve the use case goals are specified in the extensions section of the textual descriptions. No exceptions are specified at this point.

## 5.2 Describing Exceptional Behaviour

When the functional specifications of the system are stable, the discovery of exceptional functionality can begin. Exceptional functionality

refers to features of the system that complement and support the primary functionality when exceptional situations are encountered. This process proceeds by analyzing the system at three levels, system level, use case level and interaction step level. These are concerned with the exceptional situations that cause the system, user goals and interaction steps to fail, respectively.

At each level, the possible exceptional situations are first sought. The discovered exceptional situations are evaluated by their effects to each use case. Then, based on the expectations and needs of the actors, the exceptional situations are designated as actor-signalled or system-detected exceptions or both. Entries are made in the exception table to record all this information. For actor-signalled exceptions, user-goal level handlers are specified in a use case description and associated with the triggering actors and the use cases they interrupt. For system-detected exceptions, sub-function level handlers are specified in a use case description and associated to the use cases they interrupt. The handlers and new actors are added to the use case diagram accordingly.

Usually, mappings between exceptional situations, exceptions, and handlers start as one-to-one. However, after specifying the exceptional behaviour for a level, the exceptions and handlers can be refined. More specifically, common effects to the system shared by different exceptional situations can be represented by a single exception. Similarly, one handler can be generalized and used for multiple exceptions.

When the exceptional behaviour is stable and adequately refined, the process of specifying exceptional behaviour is recursively applied to all new handler use cases. Each new handler is evaluated for how every discovered exceptional situation will affect it. As a result, additional handlers are specified and existing ones are reused to interrupt and perform exceptional behaviour on exceptional behaviour.

### 5.3 Discovering Exceptional Situations

When looking for exceptional situations at the system level, we are looking for anything that will break the system as a whole or cause an actor to deviate from their goal. Interesting things to consider include: the operational needs of the system, e.g. power source, accessibility,

connectivity; and anything that will draw away the attention of an actor, e.g. emergencies, safety concerns, malicious behaviour.

At the use case level, we look at how the use case fails as a whole, without considering the failure of individual interactions contained within. For example, *BuySomething* is a use case that specifies how an actor makes purchases at an online store. At the use case level, *BuySomething* fails because the item was out of stock, or because the actor didn't have enough money. The use case would not fail at the use case level because the "add to cart" button is not properly linked (it fails at the interaction step level). The pre-conditions, post-conditions, and invariants of a use case are a good place to start looking for exceptions at the use case level.

Finally at the interaction step level, each interaction step for every use case is examined and classified into input and output interactions. Inputs and outputs may fail, so the consequences and ways to deal with such a failure must be identified. If the consequences endanger the success of the use case, then the failure must be detected and addressed by the system.

Omission of input can lead to use case failure, so for input interaction any omission needs to be addressed. For instance, prompting for the input again after a set time has elapsed, or using default inputs are possible options. Safety considerations might make it even necessary to shutdown the system in case of missing input. Invalid input is another example of an input problem that can cause use case failure. Since most actors are aware of the importance of their input, a reliable system should acknowledge reception of input and provide status indicators.

Whenever an output triggers a critical action of an actor, then the system must make sure that it can detect eventual communication problems or failure of an actor to execute the requested action. For example, an elevator's control software might tell the motor to stop, but a communication failure or misbehaviour might keep the motor going. For such critical errors, additional hardware, e.g. a motion sensor, may be necessary to ensure reliability.

As we move down from the system level to the interaction step level, there will be less and less actor-signalled exceptions found for a

couple of reasons. First, some exceptional situations found at a lower level are often already addressed by exceptions found at a higher level, thus, defining a corresponding exception and handler would be redundant. Second, failure of an interaction step is not as relevant to an actor, but failure of a user-goal level use case or of the entire system is relevant from an actor's goal driven point of view. Therefore it's less likely an actor will deviate from their goal to initiate exceptional behaviour interaction step failure. Thus, interaction step failures that lead to use case failure should be and are usually system-detected. Hence, there are more system-detected exceptions at the interaction step level.

### 5.4 Process Summary

The exception-aware process first describes all the normal behaviour of the system and then describes the exceptional behaviour of the system at the three levels, system, use case, and interaction step. At each level the possible exceptional situations are found and the respective exceptions and handlers are defined and possibly refined. The process is then applied recursively until no new exceptional behaviour is required.

In the end, the approach produces extended, exception-aware use case diagrams of the system accompanied with descriptions of every use case and an exception table. The use case diagram provides a summary of the system as partitioned into normal and exceptional behaviour. The use case descriptions consolidates all normal and exceptional information for each use case, while the exception table consolidates all information for each exception. Together these documents specify how the system is expected to behave according to the actors of the system under normal and exceptional circumstances.

# CHAPTER 6: CASE STUDY: ELEVATOR CONTROL SYSTEM

This section presents a case study of a safe and reliable *Elevator Control System* (ECS) to illustrate how the exception-aware requirements elicitation process described in Chapter 5 is used to produce use cases that detail the normal and exceptional requirements of the ECS.

## 6.1 Problem Statement

The job of the development team is to implement an ECS that coordinates the hardware components of a single-cabin elevator to carry users between floors. Initially, the hardware components including the motor, the elevator doors, and the cabin location sensors, are all considered external entities (secondary actors) to the system. The developers must also decide on the additional hardware (if any) needed to meet the functional and non-functional requirements of the ECS.

## 6.2 Normal Behaviour in the ECS Case Study

There is initially only one primary actor in the ECS, the *User*. A user has only one goal with the system, and that is to take the elevator to go from one floor (source floor) to another (destination floor), which is described in the use case *TakeElevator* shown in Fig. 6.2.1

**Use Case:** TakeElevator
**Scope:** Elevator Control System
**Primary Actor:** User
**Intention:** The intention of the *User* is to take the elevator to go to a destination floor.
**Level:** User Goal
**Main Success Scenario:**
1. *User* **Call[s]Elevator**
2. *User* **Ride[s]Elevator**

**Extensions:**
1a. The cabin is already at the floor of the *User* and the door is open. *User* enters elevator; use case continues at step 2.
1b. The *User* is already inside the elevator. Use case continues at step 2

Fig. 6.2.1: Use case description for *TakeElevator*

As described in the main success scenario, the *User* first calls the elevator to his/her current floor, and then rides it to his/her destination floor.

The *CallElevator* and *RideElevator* use cases are shown in Fig.6.2.2. To call the elevator, the User pushes the up or down button to

indicate the direction he/she wishes to go and waits for the elevator to arrive. To ride the elevator the *User* enters the cabin, selects a destination floor, and waits until the elevator arrives at the destination floor, where he/she then exits the elevator.

**Use Case:** CallElevator
**Primary Actor:** User
**Intention:** *User* wants to call the elevator to the floor that he/she is currently on.
**Level:** Subfunction
**Main Success Scenario:**

1. *User* pushes button, indicating in which direction he/she wants to go.
2. System acknowledge request.
3. *System schedules* **ElevatorArrival** for the floor the *User* is currently on.

**Extensions:**

2a. The same request already exists. System ignores the request. Use case ends in success.

**Use Case:** RideElevator
**Primary Actor:** User
**Intention:** *User* wants to ride the elevator to a destination floor.
**Level:** Subfunction
**Main Success Scenario:**

1. *User* enters elevator.
2. *User* selects a destination floor.
3. System acknowledges request and closes the door.
4. System schedules **ElevatorArrival** for the destination floor.
5. *User* exits the elevator at destination floor.

**Extensions:**

1a. *User* does not enter elevator. System times out and closes door. Use case ends in failure.

2a. *User* does not select a destination floor. System times out and closes door. System processes pending requests or awaits new request. Use case ends in failure.

5a. *User* selects another destination floor. System acknowledges new request and schedules **ElevatorArrival** for the new floor. Use case continues at step 5.

Fig. 6.2.2: Use case descriptions for *CallElevator* and *RideElevator*

*CallElevator* and *RideElevator* both include the *ElevatorArrival* use case shown in Fig. 6.2.3. It describes how the ECS directs the elevator to a specific floor. Once the system detects that the elevator is approaching the destination floor, it requests the motor to stop and then opens the door.

**Use Case:** ElevatorArrival
**Primary Actor:** N/A
**Intention:** System wants to move the elevator to a specific floor.
**Level:** Subfunction
**Main Success Scenario:**

1. System detects elevator is approaching destination floor.
2. System requests motor to stop.
3. System detects elevator is stopped at destination floor.
4. System opens door.

Fig. 6.2.3: Use Case Description for *ElevatorArrival*

The use cases that describe the normal interaction between the user and the ECS can be summarized in a standard UML use case diagram as shown in Fig. 6.2.4.

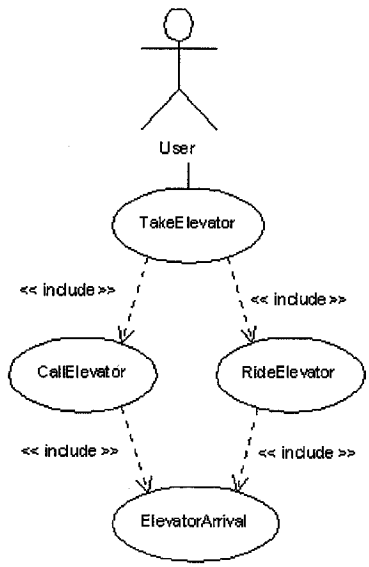


Fig. 6.2.4: ECS – normal interaction use case diagram

## 6.3 Exceptional Behaviour in the ECS Case Study

This section will specify the exceptional behaviour of the ECS by analyzing the system at three levels: system, use case, and interaction step.

### 6.3.1 System Level Exceptions

At the system level, we uncover the exceptional situations, *power failure in building* and *maintenance/repairs on elevator*, by examining the

operational requirements of the elevator's hardware. Another exceptional situation, *fire in the building* is found by considering all events in the environment that causes a user to change their goal with about using the elevator.

When a power failure occurs, the whole system stops due to lack of electricity. This exceptional situation is ignored (at least at this point) because loss of pending requests is acceptable, and employing a backup power supply is costly and impractical. Even though this exceptional situation is ignored by the system, like all exceptional situations, it is still entered in the exception table for future reference.

Maintenance and repairs are needed on a regular basis to keep the elevator's mechanical components functioning reliably, and can not be ignored. Fire and similar emergencies, which threaten human safety, can not be ignored either. One solution to both situations is to provide a feature that overrides the current operation of an elevator, and brings it to and keeps it at the main floor of the building until further notice. This is described by the handler, *ReturnToMainFloor*. *ReturnToMainFloor* interrupts all four normal use cases, which simplifies to interrupting

*TakeElevator*, because *TakeElevator* includes the other three use cases. This handler requires a new actor, called *Elevator Operator*, who has special permission to activate the use case, e.g. the building manager or a service person. *ReturnToMainFloor* is described in Fig.6.3.1.1 and Fig.6.3.1.2 shows the use case diagram at the end of looking at the system level.

**Use Case:** ReturnToMainFloor <<handler>>
**Context & Exceptions:** TakeElevator {ElevatorOverride<<interrupt & fail>>}
**Primary Actor:** Elevator Operator
**Intention:** *Elevator Operator* wants to call the elevator to the Main floor.
**Level:** User Goal
**Main Success Scenario:**

1. System clears all requests and requests motor to go down.
2. System detects that elevator is approaching the Main floor and requests motor to stop.
3. System opens elevator door.

Fig. 6.3.1.1: Use case description for *ReturnToMainFloor* handler

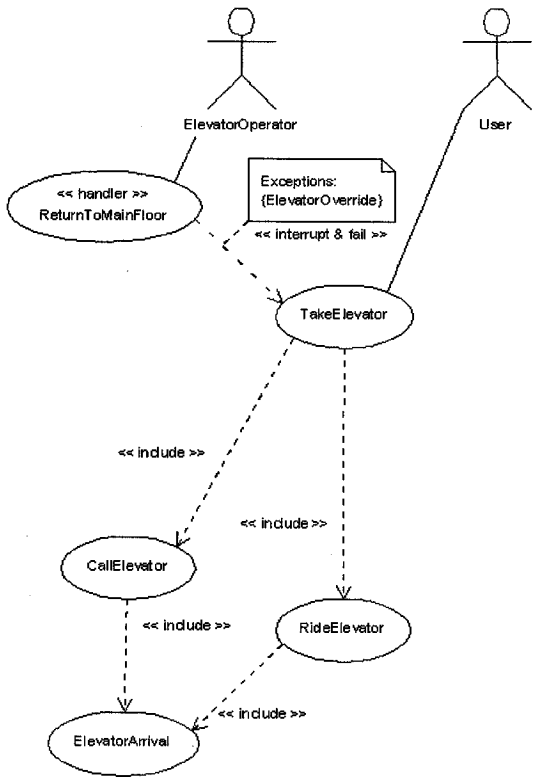


Fig. 6.3.1.2: Use case diagram at the end of system level analysis

### 6.3.2 Use Case Level Exceptions

At the use case level, we consider the following two exceptional situations, *elevator is over capacity*, and *user feels uncomfortable inside elevator*. The elevator capacity issue is found by examining the invariants of *ElevatorArrival* and *RideElevator*, which should indicate limitations to what the elevator can carry. The other exceptional situation is found by considering the safety and comfort needs of a user while inside the

elevator, should the user feel something is wrong with the elevator and wish to do something about it.

It is not reasonable to have users detect when the elevator is overweight, so the exception *ElevatorOverweight* must be system-detected. The solution is to prevent the elevator from moving when it is over capacity, and sound an alert to notify the users as captured by the handler, *OverweightAlert,* shown in Fig. 6.3.2.1. *OverweightAlert* interrupts *RideElevator*, and requires hardware that produces an audible alert and perhaps a weight sensor.

**Use Case:** OverweightAlert <<handler>>
**Context & Exceptions:** RideElevator {Overweight<<interrupt & continue>>}
**Primary Actor:** N/A
**Intention:** System wants to alert the passengers that there is too much weight in the elevator.
**Level:** Subfunction
**Main Success Scenario:**

1. System turns on the buzzer.
2. System detects that the weight is back to normal.
3. System turns off the buzzer.

Fig. 6.3.2.1: Use case descriptions for *OverweightAlert* handler

If the user feels there is something wrong with the elevator, a reasonable solution is to stop the elevator and sound an alarm that will

attract attention, which is described by the handler, *EmergencyStop* in Fig. 6.3.2.2. To ensure the elevator will stop, an emergency brake is added to the system. *EmergencyStop* can interrupt all four normal use cases, which simplifies to interrupting *TakeElevator* when the actor-signalled, *UserEmergency* is raised. Fig. 6.3.2.3 shows the use case diagram at the end of the use case level analysis.

**Use Case:** EmergencyStop <<handler>>
**Context & Exceptions:** TakeElevator {UserEmergency <<interrupt & continue>>}
**Primary Actor:** User
**Intention:** *User* wants to stop the movement of the cabin.
**Level:** User Goal
**Main Success Scenario:**

1. System stops and activates **EmergencyBrake**.
2. *User* toggles off emergency stop button.
3. System deactivates brakes and alarm, and continues processing request.

Fig. 6.3.2.2: Use case description of *EmergencyStop* handler

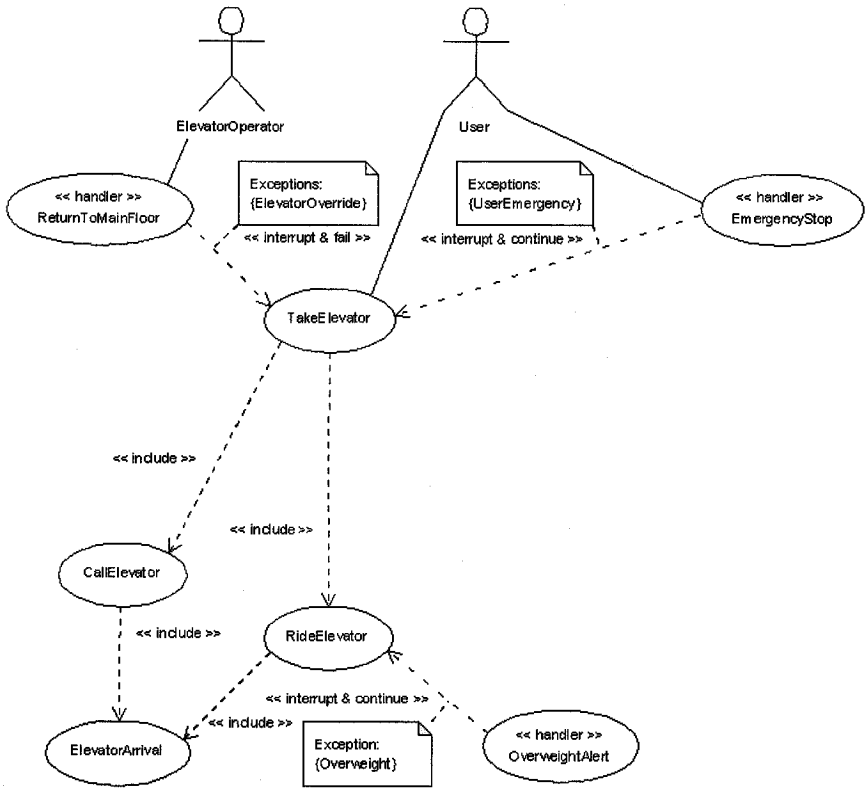


Fig. 6.3.2.3: Use case diagram at the end of use case level

Another exceptional situation, found at the use case level, occurs when *the elevator goes to the wrong floor or misses the right floor*. Arriving at the right floor is a post-condition of *ElevatorArrival*. However, this exceptional situation is ignored because it is not a critical problem. A user riding the elevator can easily rectify this by getting off at the next floor and taking the elevator or stairs back. If it is a persistent problem, then the user can activate *EmergencyStop*. Users outside the elevator can call the elevator again or give up and take the stairs.

### 6.3.3 Interaction step level exceptions

At the interaction step level, we find and address two system-detected exceptions: *MotorFailure* and *DoorStuckOpen*. We start by examining every step in ElevatorArrival. The first step involves the floor sensor informing the system that the elevator is approaching a floor. A floor sensor defect might cause the elevator to miss a destination floor. This situation is ignored because it was already addressed at a higher level. In Step 2 of *ElevatorArrival* the system requests the motor to stop. A critical exceptional situation occurs if the motor malfunctions and does not stop. So the respective handler, *EmergencyBrake*, requests the motor to stop again and activates the emergency brakes, as shown in Fig.6.3.3.1. Emergency brake is activated by the system-detected exception *MotorFailure* and is included by *EmergencyStop*, to promote the reuse of common behaviour.

**Use Case:** EmergencyBrake <<handler>>
**Context & Exceptions:** TakeElevator {MotorFailure<<interrupt & fail>>}
**Primary Actor:** N/A
**Intention:** System wants to stop operation of elevator and secure the cabin.
**Level:** Subfunction
**Main Success Scenario:**

1. System stops motor.
2. System activates the emergency brakes.

Fig. 6.3.3.1: Use case description for *EmergencyBrake* handler

In step 3 of *ElevatorArrival*, the system requests the door to open, and the door might fail to open. However, the user can always retry pressing the floor's button if inside, or call the elevator again if outside. This exceptional situation is of course more critical to a user inside the elevator, but he/she can also try another floor or worst case, activate *EmergencyStop*. So without threatening reliability, the system can ignore the exceptional situation, and hence leave it up to the user in the elevator to decide to retry the floor, go to a different floor or push the emergency button. Fig. 6.3.3.2 shows the updated version of *ElevatorArrival* that considers this scenario.

**Use Case:** ElevatorArrival
**Primary Actor:** N/A
**Intention:** System wants to move the elevator to a specific floor.
**Level:** Subfunction
**Main Success Scenario:**

1. System detects elevator is approaching destination floor.
2. System requests motor to stop.
3. System detects elevator is stopped at destination floor.
4. System opens door.

**Extensions:**

4a. Door fails to open.
System continues processing the next request (it is up to the user to select a new destination floor or press the emergency stop button). Use case ends in failure.

Fig. 6.3.3.2: Updated use case description for *ElevatorArrival*

When examining the *CallElevator* and *RideElevator* use cases, we see a common problem that can prevent the use cases from succeeding: *the elevator door gets stuck opened*, represented by the exception *DoorStuckOpen*. An obstacle or person may be preventing the door from closing. If this is the case, the response, as described in the handler, *DoorAlert* is to activate an audible alert, so whoever is blocking the door might cease to do so; Fig. 6.3.3.3 shows the use case description of *DoorAlert* and Fig. 6.3.3.4 shows the use case diagram at the end of the interaction step level analysis.

**Use Case:** DoorAlert <<handler>>
**Context & Exceptions:** TakeElevator {DoorStuckOpen<<interrupt & continue>>}
**Primary Actor:** N/A
**Intention:** System wants to alert the passengers that there is an obstacle preventing the door from closing.
**Level:** Subfunction
**Main Success Scenario:**

1. System turns on the buzzer.
2. System requests the door to close.
3. System detects that the door is now closed.
4. System turns off the buzzer.

Fig. 6.3.3.3: Use case description for *DoorAlert* handler

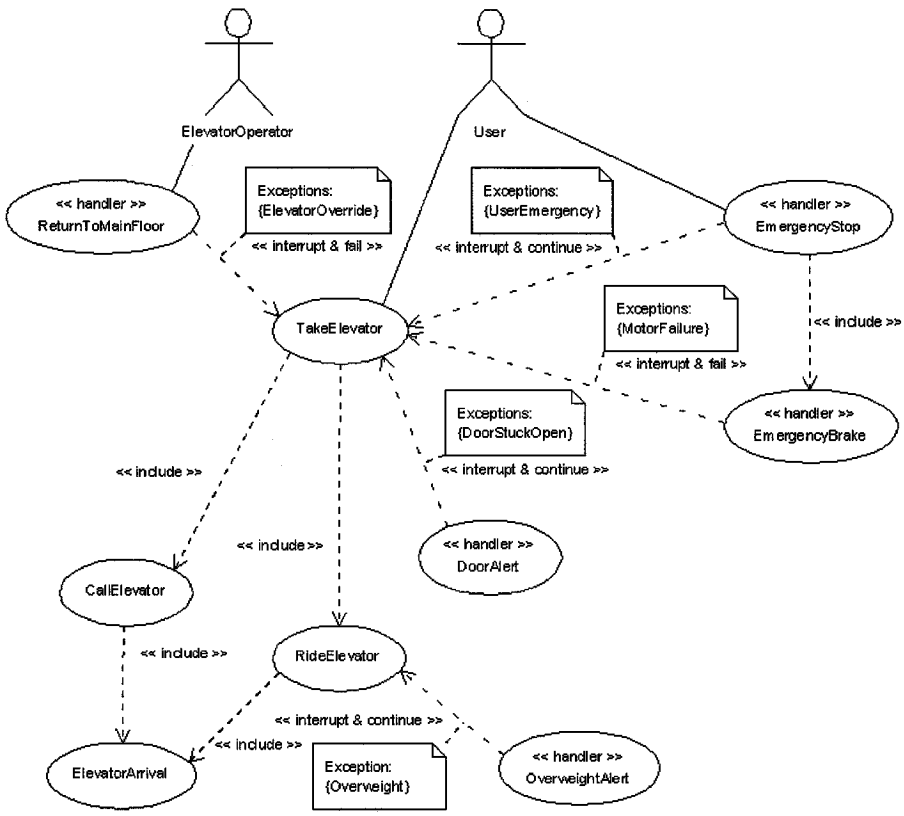


Fig. 6.3.3.4: Use case diagram for *DoorAlert* handler

The exceptional analysis of the use cases is now recursively applied to all the handlers, because handlers may be themselves, interrupted by exceptions. In our system, the *EmergencyBrake*, *OverweightAlert* and *DoorAlert* handler use cases all wait until the situation is resolved. In case the problem persists for a certain amount of time, the ECS should notify an *ElevatorOperator*. The *ElevatorOperator* can then evaluate the situation and, if necessary, call for the appropriate assistance, e.g. repairman, fire department, and/or signal an *ElevatorOverride*. This functionality is described in the handler use case *NotifyElevatorOperator* shown in Fig.6.3.3.5.

**Use Case:** NotifyElevatorOperator <<handler>>
**Context & Exceptions:**
EmergencyBrake {ElevatorStoppedTooLong <<interrupt & fail>>},
DoorAlert {DoorStuckOpenTooLong<<interrupt & fail>>},
OverweightAlert {OverweightTooLong<<interrupt & fail>>}
**Primary Actor:** N/A
**Intention:** System wants to notify the elevator operator the elevator has been stopped for too long.
**Level:** Subfunction
**Main Success Scenario:**
1. System alerts the elevator operator.

Fig. 6.3.3.5: Use case description for *NotifyElevatorOperator* handler

Another round of exceptional analysis does not uncover anymore exceptional situations so the full exception-aware use case diagram for ECS is shown in Fig.6.3.3.6. In addition, the updated and complete use case descriptions and exception table can be found in Appendix A and B, respectively.

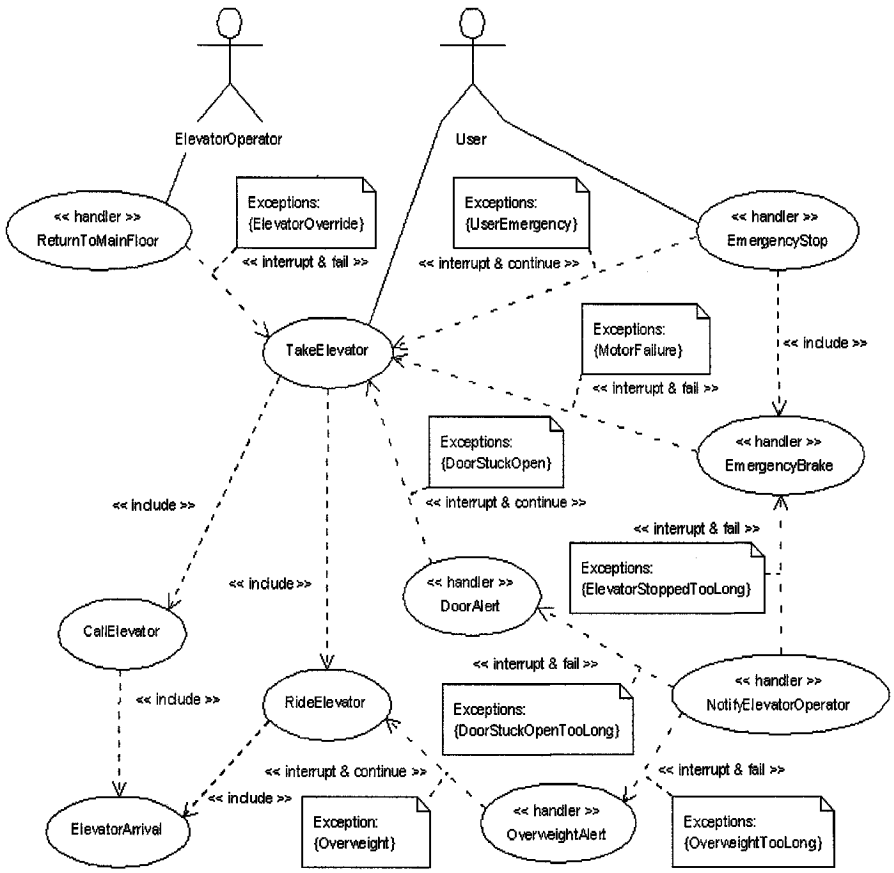


Fig. 6.3.3.6: Extended use case diagram for ECS

# CHAPTER 7: RELATED WORK

Mainstream software development methods currently deal with exceptions only at late design and implementation phases. However, several approaches have been proposed that extend exception handling ideas to other parts of the software development cycle.

De Lemos et al. [De Lemos 2001] emphasizes the separation of the treatment of requirements related, design-related, and implementation-related exceptions during the software life-cycle by specifying the exceptions and their handlers in the context where faults are identified. The description of exceptional behaviour is supported by a cooperative object-oriented approach that allows the representation of collaborative behaviour between objects at different phases of the software development.

Rubira et al. [Rubira 2004] present an approach that incorporates exceptional behaviour in the development of component-based software by extending the Catalysis software development method. The

requirements phase of Catalysis is also based on use cases, and the extension augments them with exception handling ideas.

Our approach is different from the above for several reasons. Firstly, we help the requirements engineers to discover exceptions and handlers by providing a detailed process that they can follow. Without a process, the only way a developer can discover exceptions is based on his/her imagination and experience. Secondly, our process increases reliability even more by helping the developers to detect the need for adding "feedback" and "acknowledgement" interaction steps with actors to make sure that there were no communication problems. Additionally, the process recommends adding of hardware to monitor request execution of secondary actors when necessary. Finally, our handler use cases are stand-alone, and can therefore be associated with multiple exceptions and multiple contexts.

Ryoo et al. [Ryoo 1999] proposed a process to construct system-oriented use cases from Jacobson's use cases to facilitate easier mapping to analysis phase models. Their procedure partitions normal use cases in terms of behaviour, thereby eventually transforming the original actor-

oriented view of use cases to a system-oriented view. The result reduces redundancy and expresses the requirements in a manner that is closer to analysis models. Ryoo et al. [Ryoo 1999] do not address exceptions or error handling, their dissection of use cases and actor roles to produce hierarchies for behaviour, intention, and environment coincide with this thesis' investigation of use cases. Their work is also a good starting point for exploring how use case exceptions may be mapped to analysis and design models.

Casati et al. [Casati 1999] formulated a way to model exceptional behaviour in a workflow management system through the use of activity graphs. Their work addresses how to represent exceptional situations that adversely affect a high level task in a diagram. The work takes a similar approach to this thesis and starts by defining exception terminology, e.g. detection and handling, at the concerned level of abstraction. They also propose a categorization of exceptions and triggering events, which they provide specific procedures on how each or which are represented. This thesis does not go into as detailed a categorization because use cases traditionally adopt a more black box approach to the system than activity graphs. For example, at the use case level we are usually unconcerned

with resource and process entities. Finally, their work stops at providing a formalism to represent various forms of exceptions, while this thesis also provides a process to discover exceptions.

Cysneiros et al. [Cysneiros 2001] proposed an approach to capture non-functional requirements starting at the elicitation phase of software development by integrating non-functional requirements to conceptual models, including but not limited to use cases. Instead of augmenting or extending existing models, the approach uses the LEL (Language Extended Lexicon) to build a separate perspective for the non-functional requirements which are refined and represented as graphs. The non-functional requirements are then integrated into functional requirements specified in a conceptual model in a systematic and traceable manner by linking parts of the graph to appropriate parts of the model. Because error handling also falls into non-functional requirements, their work provides an alternate means of representing error handling at the use case level.

# CHAPTER 8: FUTURE WORK

Clearly, it is necessary to explore how exceptional use cases can be mapped consistently to analysis and design models, and of course, eventually implementation exceptions. This is a daunting task because with each model, exceptions are expected to take on new meanings and behaviour as they have in this proposal. Ideally, there will be an exception-enabled model for every phase of development, which can then be checked and proved for consistency and observed for traceability.

Something that was not addressed in this thesis is the priority of interrupts. Currently, the priorities are dealt with in a naïve first-come first-serve fashion. For example, in the ECS, see Fig.6.3.3.6, if an elevator operator activates *ReturnToMainFloor* after a user activates *EmergencyStop*, *ReturnToMainFloor* is ignored and the elevator will not move. To overcome this, *ReturnToMainFloor* can be made to interrupt & fail *EmergencyStop*, thus giving priority to *EmergencyStop*. However, this system quickly breaks down and does not represent complex priorities very well.

Another issue that remains to be addressed is concurrency and multiple representations in exceptional use cases. At the system level, only one instance of each handler is used to handle an exception, but at lower levels, this is not concretely defined. In this proposal it is assumed that one instance of a handler is used for each instance of a use case, but this is not always the case. Sometimes one instance of a handler can handle all instances of a use case, and in other cases, only one instance of a use case needs to be handled not all instances. This proposal does not address how these multiplicities of relationships can be represented. As a result, it is hard to see if a handler will supersede or conflict with another handler.

It is possible to support the catch-all feature commonly found in exception enabled programming languages. At the use case level providing a catch-all feature allows developers to specify interactions required to recover from unknown exceptional situations. For example, restarting a system desktop computer sometimes serves as a fix-all/worst case remedy. Using the granularity feature of use cases, a summary use case can be defined to encapsulate a number of use cases. A general "Unknown Error" exception can then be specified to interrupt this summary

use case initiating a catch-all handler use case. However, it is not often developers will know how to handle an error without knowing the error, and this practice contradicts the definition and methodology presented in this thesis of anticipating and detecting exceptional situations. Defining such handlers can serve as a foundation to finding more exceptional situations in the later stages of software development, but further investigation is needed in later stages to see if such a feature will be useful.

Finally, in this proposal, a lot of freedom was given in how to relate exceptional situations, exceptions, handlers and interrupted use cases by allowing a many-to-many mapping between each of these elements. The problem is that this also causes a great deal of confusion because the concepts may be broken down and reused so their original meaning is lost. No method was specified in this thesis that helps developers refine these relationships and go from a basic one-to-one mapping to a more optimized many-to-many mapping. Most importantly it needs to be more extensively investigated whether all of these components need to have a many-to-many mapping. For instance, how much practical flexibility is lost if a one-to-one mapping between exceptions and handlers was enforced.

Additionally, at the use case level, an important concern is readability which favours one-to-one mappings. So a better balance between flexibility and readability needs to be found.

# CHAPTER 9: CONCLUSION

When developing reliable systems, exceptional situations that the system might be exposed to have to be discovered and addressed at the requirements elicitation phase. Exceptional situations are less common and hence the behaviour of the system in such situations is less obvious. Also, users are more likely to make mistakes when exposed to exceptional situations.

This thesis proposes an approach that extends use case based requirements elicitation with ideas from the exception handling world. A process is defined that leads a developer to systematically investigate all possible exceptional situations that the system may be exposed to, and to determine how the users of the system expect the system to react in such situations. The discovery of all exceptional situations and detailed user feedback at an early stage is essential, saves development cost, and ultimately results in a more dependable system.

It was showed how to extend UML use case diagrams to separate normal and exceptional behaviour. This allows developers to model the

handling of each exceptional situation in a separate use case, and to graphically show the dependencies among standard and handler use cases.

Based on the exception-aware use cases described, a specification that considers all exceptional situations and user expectations can be elaborated during a subsequent analysis phase. This specification can then be used to decide on the need for employing fault masking and fault tolerance techniques when designing the software architecture and during detailed design.

# APPENDIX A – USE CASE DESCRIPTIONS FOR ELEVATOR CONTROL SYSTEM

**Use Case:** CallElevator
**Primary Actor:** User
**Intention:** *User* wants to call the elevator to the floor that he/she is currently on.
**Level:** Subfunction
**Main Success Scenario:**

1. *User* pushes button, indicating in which direction he/she wants to go.
2. System acknowledge request.
3. *System schedules* **ElevatorArrival** for the floor the *User* is currently on.

**Extensions:**

2a. The same request already exists. System ignores the request. Use case ends in success.

**Use Case:** DoorAlert <<handler>>
**Context & Exceptions:** TakeElevator {DoorStuckOpen<<interrupt & continue>>}
**Primary Actor:** N/A
**Intention:** System wants to alert the passengers that there is an obstacle preventing the door from closing.
**Level:** Subfunction
**Main Success Scenario:**

1. System turns on the buzzer.
2. System requests the door to close.
3. System detects that the door is now closed.
4. System turns off the buzzer.

**Extensions:**

2a. System times out and **Notify[s]ElevatorOperator**. Use case ends in failure.

**Use Case:** ElevatorArrival
**Primary Actor:** N/A
**Intention:** System wants to move the elevator to a specific floor.
**Level:** Subfunction
**Main Success Scenario:**

7. System detects elevator is approaching destination floor.
8. System requests motor to stop.
9. System detects elevator is stopped at destination floor.
10. System opens door.

**Extensions:**

4a. Door fails to open
System continues processing the next request (it is up to the user to select a new destination floor or press the emergency stop button). Use case ends in failure.

**Use Case:** EmergencyBrake <<handler>>
**Context & Exceptions:** TakeElevator {MotorFailure<<interrupt & fail>>}
**Primary Actor:** N/A
**Intention:** System wants to stop operation of elevator and secure the cabin.
**Level:** Subfunction
**Main Success Scenario:**

1. System stops motor.
2. System activates the emergency brakes.

**Use Case:** EmergencyStop <<handler>>
**Context & Exceptions:** TakeElevator {UserEmergency <<interrupt & continue>>}
**Primary Actor:** User
**Intention:** *User* wants to stop the movement of the cabin.
**Level:** User Goal
**Main Success Scenario:**

1. System stops and activates **EmergencyBrake**.
2. *User* toggles off emergency stop button.
3. System deactivates brakes and alarm, and continues processing request.

**Extensions:**

2a. *User* does not toggle off emergency stop button. System times out and **Notify[s]ElevatorOperator**. Use case ends in failure.

**Use Case:** NotifyElevatorOperator <<handler>>
**Context & Exceptions:**
EmergencyBrake {ElevatorStoppedTooLong <<interrupt & fail>>},
DoorAlert {DoorStuckOpenTooLong<<interrupt & fail>>},
OverweightAlert {OverweightTooLong<<interrupt & fail>>}
**Primary Actor:** N/A
**Intention:** System wants to notify the elevator operator the elevator has been stopped for too long.
**Level:** Subfunction
**Main Success Scenario:**

2. System alerts the elevator operator.

**Use Case:** OverweightAlert <<handler>>
**Context & Exceptions:** RideElevator {Overweight<<interrupt & continue>>}
**Primary Actor:** N/A
**Intention:** System wants to alert the passengers that there is too much weight in the elevator.
**Level:** Subfunction
**Main Success Scenario:**

1. System turns on the buzzer.
2. System detects that the weight is back to normal.
3. System turns off the buzzer.

**Extensions:**

2a. System detects that it is still overweight. System times out and **Notify[s]ElevatorOperator**. Use case ends in failure.

**Use Case:** ReturnToMainFloor <<handler>>
**Context & Exceptions:** TakeElevator {ElevatorOverride<<interrupt & fail>>}
**Primary Actor:** Elevator Operator
**Intention:** *Elevator Operator* wants to call the elevator to the Main floor.
**Level:** User Goal
**Main Success Scenario:**

1. System clears all requests and requests motor to go down.
2. System detects that elevator is approaching the Main floor and requests motor to stop.
3. System opens elevator door.

**Use Case:** RideElevator
**Primary Actor:** User
**Intention:** *User* wants to ride the elevator to a destination floor.
**Level:** Subfunction
**Main Success Scenario:**

1. *User* enters elevator.
2. *User* selects a destination floor.
3. System acknowledges request and closes the door.
4. System schedules **ElevatorArrival** for the destination floor.
5. *User* exits the elevator at destination floor.

**Extensions:**

1a. *User* does not enter elevator. System times out and closes door. Use case ends in failure.
2a. *User* does not select a destination floor. System times out and closes door. System processes pending requests or awaits new request. Use case ends in failure.
2b. Exception {Overweight <<interrupt & continue>>}
System continues processing the requests. Use case ends in success.
4a. Exception {UserEmergency <<interrupt & continue>>}
System continues processing the next request. Use case ends in failure.
5a. *User* selects another destination floor. System acknowledges new request and schedules **ElevatorArrival** for the new floor. Use case continues at step 5.

**Use Case:** TakeElevator
**Scope:** Elevator Control System
**Primary Actor:** User
**Intention:** The intention of the *User* is to take the elevator to go to a destination floor.
**Level:** User Goal
**Main Success Scenario:**

1. *User* **Call[s]Elevator**
2. *User* **Ride[s]Elevator**

**Extensions:**

1a. The cabin is already at the floor of the *User* and the door is open.
*User* enters elevator; use case continues at step 2.
1b. The *User* is already inside the elevator. Use case continues at step 2.

# APPENDIX B – EXCEPTION TABLE FOR ELEVATOR CONTROL SYSTEM

| Exception | Exceptional Situation | Handler | Affected Use Case | Detection | Comments |
|---|---|---|---|---|---|
| Door Stuck Open | Door obstructed or broke | Door Alert | TakeElevator<br>CallElevator<br>RideElevator | System door sensor timeout | |
| Door Stuck Open Too Long | Door still obstructed | Notify Elevator Operator | OverweightAlert<br>RideElevator<br>TakeElevator | System timeout | |
| Elevator Override | Maintenance and repairs | Return To Main Floor | TakeElevator<br>CallElevator<br>RideElevator<br>ElevatorArrival | Actor, Elevator Operator | |
| Elevator Stopped Too Long | Elevator wasn't resumed | Notify Elevator Operator | EmergencyBrake<br>TakeElevator | System timeout | |
| Motor Failure | Motor does not Respond to requests | Emergency Brake | TakeElevator<br>CallElevator<br>RideElevator<br>ElevatorArrival | System floor sensor timeout | |
| N/A | Power failure | | | | Backup power too costly |
| N/A | Door won't open | | | | User can retry, not critical |
| Overweight | Elevator over capacity | Overweight Alert | RideElevator | System weight sensor hardware | |

| Exception | Exceptional Situation | Handler | Affected Use Case | Detection | Comments |
|---|---|---|---|---|---|
| Overweight Too Long | Still overweight, users aren't getting out | Notify Elevator Operator | OverweightAlert<br>RideElevator<br>TakeElevator | System timeout | |
| *See* Elevator Override | Fire, building emergency | | | | |
| User Emergency | Elevator unstable or unpredictable,<br>User in elevator uncomfortable | Emergency Stop | TakeElevator<br>CallElevator<br>RideElevator<br>ElevatorArrival | Actor, User | |

# BIBLIOGRAPHY


Casati, F., Pozzi, G.: *Modeling Exceptional Behaviors in Commercial Workflow Management Systems.* In: Fourth IECIS International Conference on Cooperative Information Systems. coopis, vol. 00, Fourth (1999) p. 127

Cockburn, A.: *Writing Effective Use Cases.* Addison–Wesley (2000)

Cysneiros, L.M., Leite, J.C.S.P, et al.: *A Framework for Integrating Non-Functional Requirements into Conceptual Models.* Requirements Engineering Journal, Vol. 6, Issue 2, Apr. (2001) pp. 97-115

De Lemos, R., Romanoysky, A.: *Exception handling in the software lifecycle.* International Journal of Computer Systems Science and Engineering 16 (2001) 167-181

Dony, C.: *Exception handling and object-oriented programming: Towards a synthesis.* In Meyrowitz, N., ed.: 4th European Conference on Object–Oriented Programming Volume 25 of ACM SIGPLAN Notices, ACM Press (1990) 322 – 330

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

Jacobson, I.: *Object-oriented development in an industrial environment.* In: Conference proceedings on Object-oriented programming systems, languages and applications, ACM Press (1987) 183 – 191

Kienzle, J., Sendall, S.: *Addressing concurrency in object-oriented software development.* Technical Report SOCS-TR-2004.8, McGill University, Montreal, Canada (2004)

Knudsen, J.L.: *Better exception-handling in block-structured systems.* IEEE Software 4 (1987) 40 – 49

Larman, C.: *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process.* 2nd edn. Prentice Hall (2002)

OMG, Object Management Group Inc.: *Unified Modeling Language: Superstructure,* Version 2.0, http://www.omg.org, (2004)

Rubira, C.M.F., de Lemos, R., Ferreira, G.R.M., Fliho, F.C.: *Exception handling in the development of dependable component-based systems.* Software — Practice & Experience 35 (2004) 195 – 236

Rumbaugh, J., Jacobson, I., Booch, G.: *The Unified Modeling Language Reference Manual.* Object Technology Series. Addison Wesley Longman, Reading, MA, USA (1999)

Ryoo, J., Stach, J.F., Park, K.: *Extension and Partitioning of Use Cases in Support of Formal Object Modeling.* In: 1999 IEEE Symposium on Application - Specific Systems and Software Engineering and Technology. asset, vol. 00, (1999) p. 238

Sendall, S., Strohmeier, A.: *Uml-based fusion analysis.* In: UML'99, Fort Collins, CO, USA, October 28-30, 1999. Number 1723 in Lecture Notes in Computer Science, Springer Verlag (1999) 278–291