Exceptional Use Cases

Aaron Shui[1], Sadaf Mustafiz[1], Jörg Kienzle[1], Christophe Dony[2]

[1]School of Computer Science, McGill University, Montreal, Canada
[2]LIRMM, Université de Montpellier, Montpellier, France
aaron@rome.com,sadaf@cs.mcgill.ca,Joerg.Kienzle@mcgill.ca,dony@lirmm.fr

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 paper 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 possible exceptional situations that the system may be exposed to: exceptional situations arising in the environment that change user goals and system-related exceptional situations that threaten to 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.

1 Introduction

Most main stream software development methods define a series of development phases – requirements elicitation, analysis, architecture and design, and finally implementation – that lead the development team to discover, specify, design and finally implement the main functionality of a system, which dictates the system's behavior most of the time. However, there are also many exceptional situations that may arise during the execution of an application. When using a standard software development process there is no guarantee that such situations are considered during the development. Whether the system can handle these situations or not depends highly on the imagination and experience of the developers. As a result, the final application might not function correctly in all possible situations.

When developing dependable systems, i.e. mission- or safety-critical systems where a malfunction can cause significant damage, nothing should be left to chance. Following the idea of integrating exception handling into the software life cycle [1,2], this paper describes an extension to standard *use case*-based requirements elicitation that leads the developers to consider all possible exceptional situations that the system under development might be exposed to. We believe that thinking about exceptional behavior has to start at the requirements phase, because it is up to the users of the system to decide how they expect the

system to react to exceptional situations. Only with exhaustive and detailed user feedback is it possible to discover and then specify the complete system behavior in a subsequent analysis phase, and decide on the need for employing fault masking and fault tolerance techniques for achieving run-time reliability during design.

The rest of the paper is structured as follows. Section 2 provides background information on use cases and exceptions. Section 3 defines some terminology and outlines the proposed extensions to standard UML use cases. Section 4 describes our proposed process, and illustrates the ideas by means of an elevator control case study. Section 5 presents related work in this area, and the last section draws some conclusions.

2 Background

2.1 Exceptions

An exceptional situation, or short *exception*, describes a situation that, if encountered, requires something exceptional to be done in order to resolve it. Hence, an *exception occurrence* during a program execution is a situation in which the standard computation cannot pursue. For the program execution to continue, an extraordinary computation is necessary [3].

A programming language or system with support for exception handling, subsequently called an *exception handling system* (EHS) [4], provides features and protocols that allow programmers to establish a communication between a piece of code which detects an exceptional situation while performing an operation (a *signaler*) and the entity or context that asked for this operation. An EHS allows users to signal *exceptions* and to define *handlers*. To *signal* an exception amounts to:

1. identify the exceptional situation,
2. to interrupt the usual processing sequence,
3. to look for a relevant handler and
4. to invoke it while passing it relevant information about the exception.

Handlers are defined on (or attached to, or associated with) entities, such as data structures, or *contexts* for one or several exceptions. According to the language, a context may be a program, a process, a procedure, a statement, an expression, etc. Handlers are invoked when an exception is signaled during the execution or the use of the associated context or nested context. To *handle* means to set the system back to a coherent state, and then:

1. to transfer control to the statement following the signaling one (*resumption model* [1]), or
2. to discard the context between the signaling statement and the one to which the handler is attached (*termination model* [1]), or
3. to signal a new exception to the enclosing context.

2.2 UML and Use Cases

The Unified Modeling Language (UML) [5] defines a *notation* for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system. UML is intentionally process-independent. However, it offers a variety of diagrams that unify the scores of graphical modeling notations that existed in the software development industry in the 80's and 90's. The diagram we are focussing on in this work is the *use case diagram*.

Since their introduction in the late 80's [6], *use cases* are a widely used formalism for discovering and recording behavioral requirements of software systems [7]. A use case describes, without revealing the details of the system's internal workings, the system's responsibilities and its interactions with its environment as it performs work in serving one or more requests that, if successfully completed, satisfy a goal of a particular stake-holder. The external entities in the environment that interact with the system are called *actors*.

In short, use cases are stories of actors using a system to *meet goals*. The standard way of achieving a goal is described in the *main success scenario*. Alternatives or situations in which the goal is not achieved are usually described in *extensions*. Use cases are in general text-based, but their strength is that they both scale up or scale down in terms of sophistication and formality, depending on the need and context. They can be effectively used as a communication means between technical as well as non-technical stake-holders of the software under development.

Use cases can be described at different levels of granularity [8]. *User-goal* level use cases describe how individual user goals are achieved. Optional *summary* level use cases provide a general overview of how the system is used. Finally, *subfunction* level use cases can be written that encapsulate subgoals of higher level use cases.

Some development methods, for example the object-oriented, UML-based development method Fondue [9], define a textual template that developers fill out when elaborating a use case. Using a predefined template forces the developer to document all important features of a use case, e.g. the *primary actor* (the one that wants to achieve the goal), the *level*, the *main success scenario* and the *extensions*. Fig. 1 shows an example Fondue use case.

Whereas individual use cases are text-based, the UML use case diagram provides a concise high level view of all (or a set of) use cases of a system. It allows the developer to graphically depict the use cases, the actors that interact with the system, and the relationships between actors and use cases.

3 Terminology and Proposed Extensions

3.1 Goals, Exceptional Goals and Failure to Achieve Goals

Each user-level use case describes a unit of useful functionality that the system under development provides to a particular actor. It details all interaction steps that an actor has to perform in order to achieve his / her goal. Typical goals

are, for instance, withdrawing money from a bank account, placing an order for a book on an online store, or using an elevator to go to a destination floor (see Fig. 1).

Sometimes, however, *exceptional situations arising in the environment*, i.e., situations that cannot be detected by the system itself, might cause actors to interact with a system in an exceptional way. The situations are exceptional in the sense that they occur only rarely, and they change the goals that actors have with the system, either temporarily or permanently. Sometimes even new actors – *exceptional actors* – start interacting with the system in case of an exceptional situation.

Very often, such situations are related to safety issues. In an elevator system, for example, a fire outbreak in the building causes the elevator operator, an exceptional actor, to activate the fire emergency mode (see Fig. 5), in which all elevator cabins go down to the lowest floor to prevent casualties or physical damage in case the ropes break. Activating the emergency behavior is an *exceptional goal* for the elevator operator, since this happens only in very rare occasions.

But even if all actors interact with the system in a normal way, *system-related exceptional situations* might prevent the system from providing the desired functionality to the actor. For example, insufficient funds can prevent a successful withdrawal, an order might not be fulfillable because the book is currently out of stock, or a motor failure might prevent a user from taking the elevator. In such cases, the goal of the actor cannot be fulfilled.

In general, the exceptional situation triggers some exceptional interaction steps with the environment. One of the steps is (or should be!) to inform the actor of the impossibility to achieve the goal. Once informed, the actor can decide how to react to the situation. The system itself might also be capable of *handling* the problem, for instance by suggesting to withdraw a smaller amount of money, or by suggesting to buy some other book, or by activating the emergency brakes and calling the elevator operator (see Fig. 6).

3.2 Exceptions in Use Cases

It is important to discover and then document all possible exceptional situations that can interrupt normal system interaction. Any exceptional situation that is not identified during requirements elicitation might potentially lead to an incomplete system specification during analysis, and ultimately in an implementation that lacks certain functionality, or even behaves in an unreliable way.

Use cases describe the interaction that happens between actors and the system under development to achieve the primary actor's goal. In Fondue, the standard way of achieving the goal is called the *main success scenario*. Use cases also offer the possibility to add extensions to the main success scenario in case the interaction takes a different route. Certain extensions can still be considered "normal" behavior, since they represent alternate ways of achieving the actor's goal.

Exceptional situations in use cases are situations that *interrupt* the flow of interaction leading to the fulfillment of the actor's goal. From now on, we'll use

the word *exception* to refer to such exceptional situations[1]. An exception occurrence endangers the completion of the actor's goal, suspending the normal interaction temporarily or for good. We propose to give names to all exceptions that can occur while interacting with the system under development, and to document them in a table together with a small textual description. As mentioned in section 3.1 we will distinguish between exceptions arising in the environment, subsequently called *actor-signaled exceptions*, and exceptions internal to the system that prevent the system from providing the requested service, subsequently called *system-detected exceptions*.

3.3 Handler Use Cases

Just like it is possible to encapsulate several steps of normal interaction in a separate subfunction-level use case, an exceptional interaction that requires several steps of handling can be described separately from the normal system behavior in a *handler use case*. The major advantage of doing this is that from the very beginning, exceptional interaction and behavior is clearly identified and separated from the normal behavior of the system. This distinction is even more interesting if it can be extracted at a glance from the use case diagram.

In a use case diagram, standard use cases appear as ellipses (see Fig. 10), associated to the actors whose goals they describe. We propose to identify handler use cases with a `<<handler>>` stereotype, which differentiates them from the standard use cases. To allow developers to identify exceptional behavior at a glance, handler use cases can be represented in the use case diagram with a special symbol or using a different color. Handler use cases for actor-signaled exceptions, i.e. handlers that describe exceptional goals, are self-contained, just like standard use cases. Handlers that address system-detected exceptions on the other hand may not necessarily be meaningful by themselves, but only within the context of a normal use case. However, handlers are full-fledged use cases in the sense that they can include sub-level handler use cases, or have themselves associated handlers that address exceptions that might occur during the processing of an exception.

Separation of handlers also enables subsequent reuse of handlers. Just like a subfunction-level use case can encapsulate a subgoal that is part of several user goals, a handler use case can encapsulate a common way of handling exceptions that might occur while processing different user goals. Sometimes even, different exceptions can be handled in the same way. Associating handler use cases to other use cases is described in section 3.4.

3.4 Linking Exception, Handlers and Use Cases

Just like in standard exception handling, where exception handlers are associated to exception handling contexts, handler use cases apply to a base use case, in

[1] It is important to point out that the meaning of exception at the requirements level is *not* directly related to exceptions as defined by modern programming languages. The term exception is used at a higher level of abstraction here.

this case any standard use case or other handler use case. We suggest to depict this association in the use case diagram by a directed relationship (dotted arrow) linking the handler use case to its base use case.

This relationship is very similar to the standard UML `<<extends>>` relationship. It specifies that the behavior of the base use case may be affected by the behavior of the handler use case in case an exception is encountered. Similar to the explicit extension points introduced in UML 2.0, the base use case can specify the specific steps in which the exception might occur (see Fig. 7 step 4a), but does not need to. In the latter case, the exceptional situation can affect the base processing at any time.

In case of an occurrence of an exceptional situation, the base behavior is put on hold or abandoned, and the interaction specified in the handler is started. A handler can temporarily take over the system interaction, for instance to perform some compensation activity, and then switch back to the normal interaction scenario. In this case, the relationship is tagged with a `<<interrupt & continue>>` stereotype. Some exceptional situations, however, cannot be handled smoothly, and cause the current goal to fail. Such dependencies are tagged with `<<interrupt & fail>>`. This is similar to the resumption and termination models reviewed in section 2.1.

The `<<interrupt & continue>>` and `<<interrupt & fail>>` relationships also differ from the `<<extends>>` relationship in the sense that they apply also to all sub use cases of a base use case. In the elevator example presented in the next section, for instance, an *Emergency Override* can interrupt *Take Elevator*, and therefore also any of the included use cases of *Take Elevator*, namely *Call Elevator*, *Ride Elevator* and *Elevator Arrival*.

Finally, the exceptions that activate the handler use case are added to the interrupt relationship in a UML comment. The notation follows the notation that was introduced in UML 2.0 to specify extension points for use cases. An example of an extended use case diagram for the elevator system with all exceptions, handler use cases and relationships is shown in Fig. 10.

4 Exception-Aware Process and Elevator Case Study

This section introduces our exception-aware requirements elicitation process and illustrates it based on a case study, a reliable and safe elevator system. For the sake of simplicity, there is only one elevator cabin that travels between the floors. There are two buttons on each floor (except for the top and ground floors) to call the lift, one for going up, one for going down. Inside the elevator cabin, there is a series of buttons, one for each floor.

The job of the development team is to decide on the required hardware, and to implement the elevator control software that processes the user requests and coordinates the different hardware devices. So far, only "mandatory" elevator hardware has been added to the system. The approaching of the cabin at a floor is detected by a sensor. The elevator control software may ask the motor to go up, go down or stop, and the cabin door to open or close.

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
Frequency & Multiplicity: A *User* can only take one elevator at a time. However, several *Users* may take the elevator simultaneously.
Main Success Scenario:
1. User **CallsElevator**.
2. User **RidesElevator**.

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. 1. TakeElevator Use Case

4.1 Describing Normal Interaction

To start off the requirements elicitation phase, the use cases that describe the interaction with the system under *normal* conditions are elaborated. In the elevator system there is initially only one primary actor, the *User*. A user has only one goal with the system, namely to take the elevator to go to a destination floor, described in the user-goal level use case *TakeElevator* shown in Fig. 1.

As we can see from the main success scenario, the *User* first calls the elevator (step 1), and then rides it to the destination floor (step 2). The potential concurrent use of the elevator is documented in the *Frequency & Multiplicity* section [10].

The *CallElevator* and *RideElevator* use cases are shown in Fig. 2. To call the elevator the *User* pushes the up or down button and waits for the elevator cabin to arrive. To ride the elevator the *User* enters the cabin, selects a destination floor, waits until the cabin arrives at the destination floor and finally exits the elevator.

CallElevator and *RideElevator* both include the *Elevator Arrival* use case shown in Fig. 3. It is a subfunction level use case that describes how the system 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 opens the door.

The use cases that describe the normal interaction between the user and the elevator control system can be summarized in a standard UML use case diagram as shown in Fig. 4.

4.2 Actor-Signaled Exceptions

The next step in our process consists in identifying exceptional situations that arise in the environment that make actors deviate from their initial goal, or change their goals completely. Sometimes actors change their goals spontaneously, sometimes changes in the environment influence the behavior of actors. In any

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 acknowledges 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: Ride Elevator
Primary Actor: User
Intention: The *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 requests. 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. 2. *CallElevator* and *RideElevator* Use Case

Use Case: ElevatorArrival
Primary Actor: N/A
Intention: System wants to move the elevator to the *User*'s destination floor.
Level: Subfunction
Main Success Scenario:
1. System detects elevator is approaching destination floor.
2. System requests motor to stop.
3. System opens door.

Fig. 3. *ElevatorArrival* Use Case

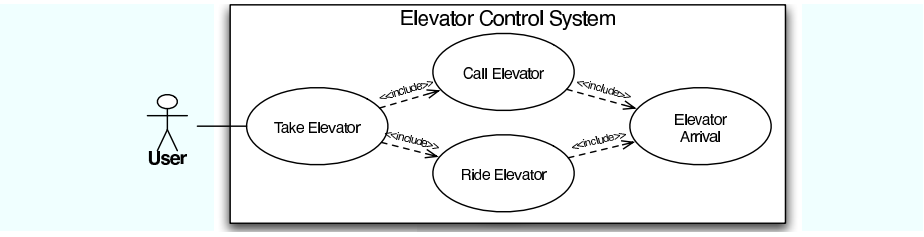

Fig. 4. Standard Elevator Use Case Diagram

Use Case: UserEmergency `<<handler>>`
Contexts & Exceptions: TakeElevator{EmergencyStop}
Primary Actor: User
Intention: *User* wants to stop the movement of the cabin.
Level: User Goal
Frequency & Multiplicity: Since there is only one elevator cabin, only one *User* can activate the emergency at a given time.
Main Success Scenario:
1. System initiates **EmergencyBrake**.
2. *User* toggles off emergency stop button.
3. System deactivates brakes and continues processing requests.

Use Case: ReturnToGroundFloor `<<handler>>`
Contexts & Exceptions: TakeElevator{EmergencyOverride}
Primary Actor: Elevator Operator
Intention: Elevator Operator wants to call the elevator to the ground floor because the elevator operation is too dangerous.
Level: User Goal
Frequency & Multiplicity: Only one ReturnToGroundFloor use case can be active at a given time.
Main Success Scenario:
1. System clears all requests and requests motor to go down.
2. System detects that elevator is approaching the ground floor and requests motor to stop.
3. System opens elevator door.

Fig. 5. *UserEmergency* and *ReturnToGroundFloor* Handler Use Case

case, the system has to interrupt its current processing and try to fulfill the new goal.

Exceptions arising in the environment are communicated to the system by special actions of actors – hence their name *actor-signaled exceptions*. A dependable system must react to actor-signaled exceptions in a well-specified way. If the handling requires exceptional interaction steps with the primary actor or other secondary actors, then a handler use case must be defined. The handler is then linked to the context, i.e. the use case in which th exception can occur.

Actor-signaled Exceptions in the Elevator Case Study In the elevator case study we identified two actor-signaled exceptions. *EmergencyStop* is signaled by the *User* actor pushing the emergency button in the elevator in case he wants to interrupt the movement of the cabin. *EmergencyOverride* is signaled by an exceptional actor, the elevator operator, using the emergency override key on the ground floor in case of an emergency, for example a fire outbreak in the building. In our case, both exceptions can interrupt the normal system operation at any time, so their context is *TakeElevator*.

Fig. 5 shows the handler *UserEmergency* that handles the exception *EmergencyStop*. The system immediately activates the emergency brakes. Subsequently, the *User* can toggle off the emergency button to reactivate the elevator. The system then resumes the original use case because the relation between *TakeElevator* and *UserEmergency* is `<<interrupt & continue>>`.

The *EmergencyOverride* exception is handled by the *ReturnToGroundFloor* handler use case, also shown in Fig. 5. *ReturnToGroundFloor* interrupts and fails the *TakeElevator* use case.

4.3 System-Detected Exceptions

Each use case must now be examined to see if there are any system-related exceptional situations that can make the use case goal fail. Up to now we have assumed that actors are reliable, that hardware never fails, and that communication with hardware and actors is reliable as well. However, this is an unrealistic assumption that a safety-critical application such as the elevator control software cannot make.

Each use case must be looked at step by step, and every interaction classified into *input* and *output* interactions. Inputs and outputs may fail, and the consequences and ways of dealing with such a failure must be identified. If the consequences endanger the accomplishment of the user goal, then the system must detect the failure – hence the name *system-detected exception* – and address the situation. Detection might require additional hardware or timeouts.

Once the exception is detected, ways of addressing the exception have to be investigated. Very often, actors – especially humans – are "surprised" when they encounter an exceptional situation, and are subsequently more likely to make mistakes when interacting with the system. Exceptional interactions during exception handling must therefore be as intuitive as possible, and respect the actor's needs. Again, all interaction steps addressing an exception have to be recorded in handler use cases.

Input Problems If omission of input from an actor can cause the goal to fail, then, once the omission has been detected, different options of handling the situation have to be considered. For instance, prompting the actor for the input again after a given time has elapsed, or using default input are possible options. Safety considerations might make it even necessary to temporarily shutdown the system in case of missing input. Invalid input data is another example of input problem that might cause the goal to fail. Since most of the time the actors are aware of the importance of their input, a reliable system should also acknowledge input from an actor, so that the actor realizes that she is making progress in achieving her goal.

Output Problems 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, the elevator control software might tell the motor to stop, but a communication failure or a motor misbehavior might keep the motor going. Again, additional hardware, for instance, a sensor that detects when the cabin stopped at a floor, or timeouts might be necessary to ensure reliability.

Use Case: RedirectElevator `<<handler>>`
Context & Exception: ElevatorArrival{MissedFloor}
Primary Actor: N/A
Intention: System redirects the elevator to a different floor because the destination floor is unreachable.
Level: Subfunction
Main Success Scenario:
1. System cancels request to stop at destination floor.
2. System detects elevator is approaching a floor.
3. System requests the motor to stop.
4. System detects elevator is stopped at floor.

Use Case: EmergencyBrake `<<handler>>`
Context & Exception: TakeElevator{MotorFailure}
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.
2. System turns on the emergency display.

Fig. 6. *RedirectElevator* and *EmergencyBrake* Handler Use Case

System-Detected Exceptions in the Elevator Case Study To illustrate the process, let us go step by step through the use case *ElevatorArrival* (see Fig. 3). 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. In this case, the corresponding handler *RedirectElevator*, shown in Fig. 6, stops the cabin at the next floor.

In Step 2 of *ElevatorArrival* the system requests the motor to stop. In case the motor malfunctions and does not stop, the emergency brakes have to be activated immediately. This is done by the *EmergencyBrake* handler, also shown in Fig. 6.

Finally, in step 3 of *ElevatorArrival*, the system requests the door to open. This output can only be sent *after* a successful stop of the motor. For reliability reasons, a "stop detection" mechanism, such as an additional sensor that monitors the cabin speed, must be added to the system. Additionally the door might fail to open in step 3. In this case, the elevator could move to a different floor and try to open the door there. Without threatening reliability, we can also choose to ignore the failure and continue processing the next request, and hence leave it up to the user in the elevator to decide to either retry the floor, go to a different floor or push the emergency button. Fig. 7 shows the updated, reliable version of the *ElevatorArrival* use case.

Looking at the *CallElevator* and *RideElevator* use case, we can detect a common problem that might prevent the goals from succeeding: the elevator door might be stuck open, for instance because an obstacle prevents it from closing. This case is handled by the *DoorAlert* handler use case. Another exceptional situation occurs when there are too many passengers in the elevator.

Use Case: ElevatorArrival
Primary Actor: N/A
Intention: System wants to move the elevator to the *User*'s destination 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. Exception{DoorStuckClosed}
System continues processing the next request (it is up to the user to select a new destination floor or press the emergency button). Use case ends in failure.

Fig. 7. Updated *ElevatorArrival* Use Case

Use Case: DoorAlert `<<handler>>`
Primary Actor: N/A
Context & Exception: TakeElevator{DoorStuckOpen}
Intention: System wants to alert the passengers that there is an obstacle preventing the door from closing.
Level: Subfunction
Main Success Scenario:
1. System displays "door open".
2. System turns on the buzzer.
3. System requests the door to close.
Step 3 is repeated until the door closes.
4. System detects that the door is now closed.
5. System turns off the buzzer.
6. System clears the display.

Use Case: OverweightAlert `<<handler>>`
Primary Actor: N/A
Context & Exception: RideElevator{Overweight}
Intention: System wants to alert the passengers that there is too much weight in the elevator.
Level: Subfunction
Main Success Scenario:
1. System displays "overweight".
2. System turns on the buzzer.
3. System detects that the weight is back to normal.
4. System turns off buzzer.
5. System clears display.

Fig. 8. *DoorAlert* and *OverweightAlert* Handler Use Case

The *OverweightAlert* handler addresses this exception. The *DoorAlert* and *OverweightAlert* handlers are shown in Fig. 8.

The step-by-step analysis of the use cases must then be recursively applied to all the handlers, because handlers may themselves be 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 elevator control system should notify an elevator

Use Case: CallElevatorOperator `<<handler>>`
Context & Exception: EmergencyBrake{ElevatorStoppedTooLong], OverweightAlert{OverweightTooLong}, DoorAlert{DoorStuckOpenTooLong}
Intention: The system wants to alert the elevator operator, so that the elevator operator can come and assess the damage.
Level: Subfunction
Main Success Scenario:
1. System cancels all pending requests.
2. System displays "calling operator ".
3. System calls operator.

Fig. 9. *CallElevatorOperator* Handler Use Case

operator. The elevator operator can then evaluate the situation and, if necessary, call a service person. This functionality is described in the handler use case *CallElevatorOperator* shown in Fig. 9.

4.4 Requirements Elicitation Summary

In parallel to the elaboration of the individual use cases and handlers, we propose to build an extended exception-aware use case diagram providing a detailed and precise summary of the partitioning of the system into normal and exceptional interactions. The diagram follows the syntax described in sections 3.3 and 3.4. User expectations of handling exceptional situations are documented in handler use cases identified with the `<<handler>>` stereotype, and attached to their respective contexts with `<<interrupt & continue>>` or `<<interrupt & fail>>` relationships. For traceability and documentation reasons, the diagram should also be accompanied with a table that records all discovered exceptions, together with a small textual description of the situation, the exception context, the associated handler, and the mechanism of detecting the situation.

The extended use case diagram for the elevator control system is shown in Fig. 10. The aforementioned exception table for the elevator system with the detailed descriptions of each exception is not shown here for space reasons.

5 Related Work

Main stream 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. [2] emphasize 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 behavior is supported by a cooperative object-oriented approach that allows the representation of collaborative behavior between objects at different phases of the software development.

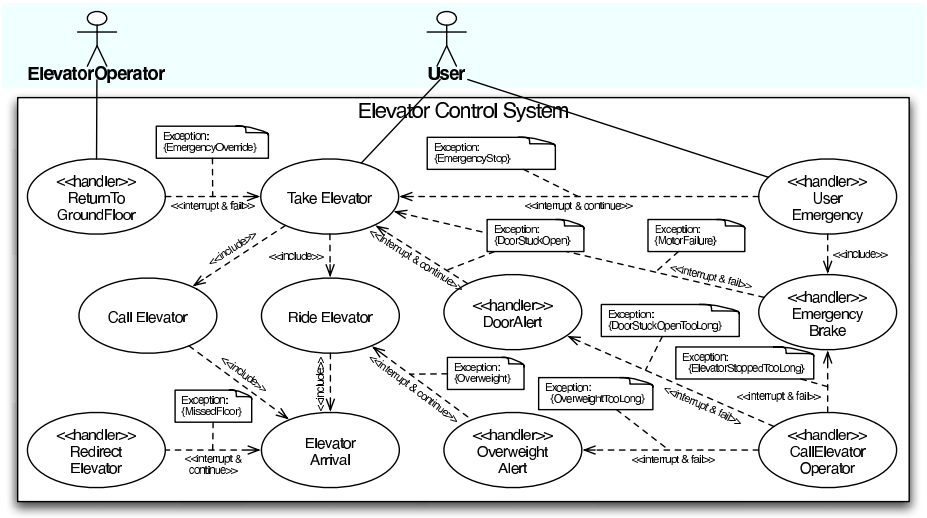

Fig. 10. Reliable Elevator Use Case Diagram

Rubira et al. [11] present an approach that incorporates exceptional behavior 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 her imagination and experience. Secondly, our process increases reliability even more by helping the developers 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.

6 Conclusion

We believe that 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 behavior of the system in such situations is less obvious. Also, users are more likely to make mistakes when exposed to exceptional situations.

In this paper we propose an approach that extends use case based requirements elicitation with ideas from the exception handling world. We define a process 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.

We also show how to extend UML use case diagrams to separate normal and exceptional behavior. 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 our exception-aware use cases, 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.

For more information on our exception-aware process, and for details on how we extended the UML 2.0 metamodel to incorporate our extensions, the interested reader is referred to [12].

References

1. Goodenough, J.B.: Exception handling: Issues and a proposed notation. Communications of the ACM **18** (1975) 683 – 696
2. de Lemos, R., Romanovsky, A.: Exception handling in the software lifecycle. International Journal of Computer Systems Science and Engineering **16** (2001) 167 – 181
3. Knudsen, J.L.: Better exception-handling in block-structured systems. IEEE Software **4** (1987) 40 – 49
4. Dony, C.: Exception handling and object-oriented programming: Towards a synthesis. In Meyrowitz, N., ed.: 4th European Conference on Object–Oriented Programming (ECOOP '90). ACM SIGPLAN Notices, (ACM Press)
5. Object Management Group: Unified Modeling Language: Superstructure. (2004)
6. 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
7. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. 2nd edn. Prentice Hall (2002)
8. Cockburn, A.: Writing Effective Use Cases. Addison–Wesley (2000)
9. 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
10. Kienzle, J., Sendall, S.: Addressing concurrency in object-oriented software development. Technical Report SOCS-TR-2004.8, McGill University, Montreal, Canada (2004)
11. 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
12. Shui, A.: Exceptional use cases - Master Thesis, School of Computer Science, McGill‘ University (2005)