On Exceptions and the Software Development Life Cycle

Jörg Kienzle
School of Computer Science, McGill University
Montreal, QC H3A2A7, Canada
Joerg.Kienzle@mcgill.ca

This paper presents the insights we gained in our research aimed at integrating exceptions and exception handling into the entire software development life cycle. We argue that exceptions are of different nature depending on the level of abstraction that the system under development is looked at. We outline a mapping relating exceptions at a high level of abstraction to exceptions and other software artifacts at lower levels of abstraction, and show that some exceptions introduced at a low level of abstraction also require the definition of corresponding exceptions at a higher level of abstraction in the case where transparent handling at the low level is not possible. Finally, we list the potential benefits of integrating exception handling into the entire software development life cycle.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Life cycle*; D.2.10 [**Software Engineering**]: Design—*Methodologies*

Keywords

exceptions, exception handling, software development life cycle, requirements, software architecture, design, implementation

1. INTRODUCTION

Exceptions originated at the programming language level, and although the ideas of exceptions and exception handling have been known for over 40 years, these concepts are currently not an integral part of standard software development methodologies. Exceptions are mostly used during the implementation phase, if at all, and even then there are often no clear guidelines of when and how to use them.

Object-orientation also originated at the programming language level. Nowadays, however, object-orientation is applied at all phases of software development. An object is always an entity that has it's own identity, and encapsulates state and behavior. However, the notion of *object* has slightly different meanings depending on the development phase in which it is used. At the *requirements engineering* phase, objects represent domain concepts, e.g., external actors that interact with the system under development, or entities encapsulating state which allows the system to reason about the problem domain. Objects at the *software architecture level* usually represent components that partition the system into modules that interact at run-time through well-defined interfaces. Sometimes, the partitioning into components can also be used to distribute the application onto multiple machines. At the *design phase*, a solution implementing the specified problem is designed. To this aim, objects are created with well-defined responsibilities and state. Algorithms to achieve the desired behavior are devised, and are realized by interacting objects. Finally, at the *implementation phase*, the individual objects are implemented by elaborating the program statements that achieve the desired functionality for each operation or method.

Using the concept of object throughout the entire software development process is very beneficial. Although the meaning of an object changes throughout the development activities, object-orientation provides traceability from the requirements phase down to the implementation phase by relating objects at different levels. Of course, the mapping between objects from one phase to the other is not always one-to-one. Often, a requirements-level object is realized using many design objects. Furthermore, during software architecture and detailed design, new objects are added that realize a specific solution. Nevertheless, object-orientation provides a unified structuring methodology that has proven to streamline software development considerably.

We and others [1, 10] believe that integrating exceptions into the entire software development process can provide similar advantages. This paper presents the insights we gained over the last few years in our research efforts that were aimed towards achieving this goal.

The rest of the paper is structured as follows: section 2 presents background on exceptions and exception handling, on the idealized fault-tolerant component and on exceptions within the software development life cycle; section 3 looks at the nature of exceptions and handlers at various phases of software development; section 4 presents the mapping we established between exceptions at the different levels of abstraction; section 5 outlines the expected benefits of integrating exceptions into the software development life cycle, and the last section draws some conclusions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WEH '08, November 14, Atlanta, Georgia, USA
Copyright 2008 ACM 978-1-60558-229-0 ...$5.00.

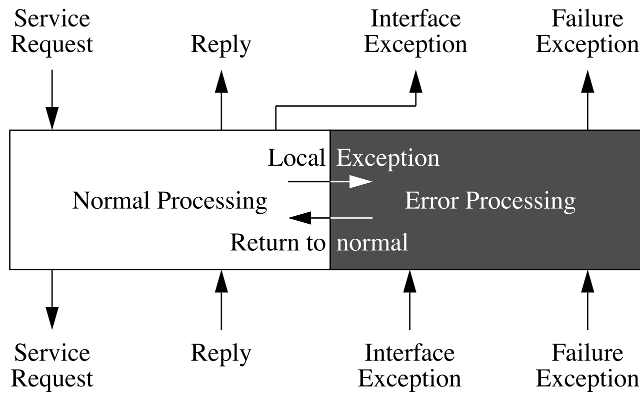

Figure 1: The Idealized Fault-Tolerant Component

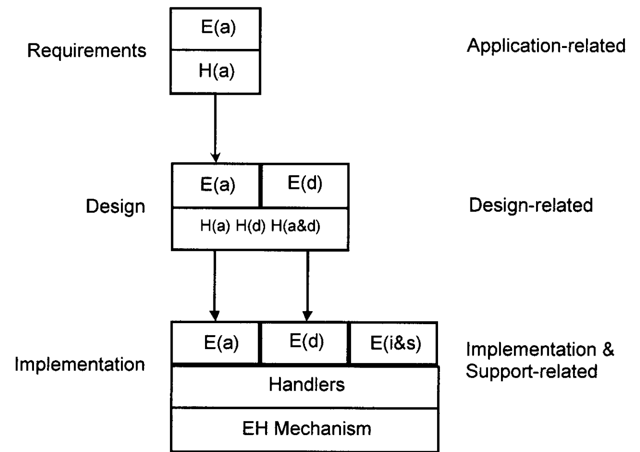

Figure 2: Exceptions in the Software Life Cycle

2. EXCEPTIONS

2.1 Background on Exceptions and Handlers

An *exception* describes a situation that, if encountered, requires something exceptional to be done in order to resolve it. During program execution, an *exception occurrence* is a situation in which the standard computation cannot pursue. For the program execution to continue, an extraordinary computation is necessary [7].

A programming language or system with support for exception handling allows users to *signal exceptions* and to define *handlers* [2]. To signal an exception amounts to detecting the exceptional situation, interrupting the usual processing sequence, looking for a relevant handler, and then invoking it.

Handlers are defined on (or attached to) 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 of the use of the associated context or nested context. To handle means to put the system into a coherent state, i.e. to carry out forward error recovery, and then to take one of these steps: transfer control to the statement following the signaling one (*resumption* model [4]), or discard the context between the signaling statement and the one to which the handler is attached (*termination* model [4]), or signal a new exception to the enclosing context.

2.2 Idealized Fault-Tolerant Component

The *Idealized Fault-Tolerant Component* (IFTC) [8] is a concept that was developed to structure the execution of dependable software (see Fig. 1). An IFTC offers services that may return replies to the component that made a service request. If a request is malformed, the component signals this by raising an *interface exception*, otherwise it executes the request and produces a reply. If an internal exception signaling an error occurs, error processing is activated in an attempt to handle the error. If it can be dealt with, normal processing in the component resumes; if not, the component itself signals its failure by an exception.

2.3 Exceptions and Software Development

Lemos and Romanovsky [1] describe the general idea of integrating exceptions into the software development life cycle using the figure shown in Fig. 2.

For each identified phase of the software development life cycle, a class of exceptions is defined. For instance, during the requirements phase, the authors propose to identify the exceptions related to the application (E(a)), and to define application-specific handlers. The authors give as an example the case of an Internet book store: if a client is looking for a book that is currently unavailable (application-level exception), the bookstore could return a list of other books from the same author, or inform about books on the same topic (application-specific handling).

During the design phase, the intent is to identify all exceptions related to the design (E(d)), i.e. related to the chosen software architecture or object-interactions. The design-level handlers for these exceptions should recover the application state to a known consistent state. In the Internet bookstore example, a crashed server (design-related exception) could be tolerated by routing the following requests to a backup server.

During the implementation phase, the intent is to identify all exceptions related to the implementation of the application and the support in which the application is executed (E(i&s)). Implementation-specific handling in the Internet bookstore example would consist, for instance, in executing a garbage collector when the free memory on the server is low.

Lemos and Romanovsky [1] suggest also that there might be a need for combined handlers, i.e. handlers that are designed to handle situations in which, for instance, application and design-specific exceptions have to be addressed together.

We agree with the authors of [1] that exceptions are of different nature at each phase of the software development process. We agree also that new exceptions appear during each development phase. However, we believe that the relationships among the exceptions and handlers of different phases are complex and deserve further investigation. The following section present our notion of exceptions and handlers at each phase of software development.

3. NATURE OF EXCEPTIONS AND HANDLING WITHIN EACH DEVELOPMENT PHASE

3.1 Exceptions and Requirements Elicitation

During requirements elicitation, an activity that is carried out at the very beginning of the software development cycle, the focus of the developer is on *discovering* and documenting essential functionality and behavior of a (software) system that does not yet exist. *Use case* [6] or other goal-oriented approaches are very popular for requirements elicitation, since they guide the developer in discovering essential services that the system under development needs to provide. 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 stakeholder. The external entities in the environment that interact with the system are called *actors*. The actor that initiates a use case in order to pursue a goal is called the *primary actor*, actors that the system interacts with in order to provide a service are called *secondary actors*.

During the requirements elicitation phase the developer works at a very high level of abstraction. The system is looked at as a *black box*: only the services that the system offers and the interactions that are necessary to achieve these services are investigated. In this context, we propose to define an exception as follows:

> *Proposition 1*: An exception at the requirements elicitation phase (at a level of abstraction where the system is considered a collection of services) represents any potentially occurring *exceptional situation* E_{s} that could prevent the system from providing the services it normally provides.

Based on our experience, there are two cases of exceptions at the requirements elicitation phase: 1) *context-affecting exceptions* $E_{\mathrm{s}}(ca)$ and 2) *service-related exceptions* $E_s(sr)$.

Context-affecting exceptions are situations that change the context in which the system operates. Certain context changes might require the system to suspend some of it's normal services for safety reasons, or to provide special *exceptional services* in order to satisfy stakeholder needs while the exceptional situation is in effect. These exceptional services can be seen as handlers for the exceptional situation ($H_{\mathrm{s}}(ca)$). For example, in an elevator system where safety is the main concern, in case of a fire outbreak in the building (exceptional situation), the elevator operator or a smoke detector (exceptional actors) should activate the fire emergency mode of the elevator control software. During a fire emergency, all elevator cabins are moved to the ground floor (handler).

Service-related exceptions represent exceptional situations in which the completion of a particular service or goal may be threatened. Service-related exceptions have many natures:

- The current system state makes the provision of a service impossible;
- Failure of secondary actors that are necessary for the completion of the user goal;
- Failure of communication links between the system and important secondary actors;
- Actors violate the system interaction protocol, i.e. they invoke system services in the wrong order, or at the wrong time.

If the service-related exception puts the user in danger, then measures must be taken to put the system in a safe state. If the service-related exception threatens the successful completion of the service, reliability is at stake. It should then be investigated in consultation with the stakeholders if the system can recover and meet the user goal in an alternative way or provide some form of degraded service. Handlers should then be defined that describe the interactions between the environment and the system needed to address the exceptional situation ($H_{\mathrm{s}}(sr)$).

After the occurrence of any exceptional situation, the system should evaluate if the encountered problem threatens the reliability or safety of future service provision. If yes, then a *mode switch* is necessary. Switching to a different operation mode allows the system to signal to the environment that the services offered by the system have changed, and reject any requests for services that cannot be performed with sufficient reliability or safety.

3.2 Exceptions and Requirements Specification

During requirements specification, one important activity consists in outlining the *system interface*. At this level of abstraction, the system under development is still a black box, i.e. no internal components are visible to the outside. The focus is on the messages that are sent to and from the system.

Applying the ideas of the idealized fault-tolerant component (see subsection 2.2) to the system and the actors, we propose to define exceptions at this level of abstraction as follows:

> *Proposition 2*: An exception at the *requirements specification phase* E_{m} (at a level of abstraction where the system is considered a black box without internal structure) is an *exceptional message* exchanged between the system and an actor in the environment (to signal an exceptional outcome of a service request, or to inform the system of a significant change in the environment that could affect the currently provided services), or the *absence of a normal message*, or the reception of a normal message that should not be received at this point according to the interaction protocol.

More precisely, exceptional messages can occur in 3 cases:

1. If an exceptional situation arises in the environment that threatens system safety or the reliability of certain services, the (exceptional) actor that detects the situation should notify the system with the appropriate exceptional message ($E_{\mathrm{m}}(ca)$).

2. In order to complete a service, the system might request services from secondary actors. If the secondary actor cannot provide the service, it should notify the system of the failure (or of the degraded service provision, if some partial result was achieved) ($E_{\mathrm{m}}(sr)$).

3. Likewise, when a service is requested from the system, in the case where the service cannot be provided as advertised, the system should notify the primary actor with an appropriate exceptional message ($E_{\mathrm{m}}(out)$).

Case 1 exceptions are asynchronous messages, since the exceptional situation can occur at any point in time. Case 1 exceptions signal the occurrence of a context-affecting exception (described in section 3.1) to the system. Case 2 and 3 exceptions on the other hand are synchronous messages, since the exception is generated as a result of a service request.

Ideally, every secondary actor would signal its failure to the system by sending it an exceptional message. However, it is dangerous to assume that the secondary actors in the environment and the communication channels are reliable. Hence, sometimes it is necessary to detect failures of secondary actors using, for instance, timeouts. Sometimes even the reception of a normal message can indicate the occurrence of a service-related exception. For instance, if an elevator receives a message notifying the system that the elevator cabin is approaching the 10th floor, then the system might conclude that the floor sensor located at the ninth floor is broken, if the previously received message established that the elevator cabin was approaching the eighth floor and the cabin was moving upwards.

Apart from the system interface, requirements specification also includes the specification of the conceptual system state. Parts of the system state, namely that part of the state that is needed to remember the current system mode, to keep track of the failures of secondary actors and to verify the adherence to the system protocol, can be considered exceptional as well.

As long as the architecture of the system is not known, run-time interaction between system components cannot be determined yet. Therefore, a high-level requirements specification usually describes the functionality of the system by specifying how service requests affect the conceptual system state. This can, for instance, be done using pre- and post-conditions that are attached to (atomic) system operations. The same strategy can be used to describe the functionality that should be provided by handlers.

3.3 Exceptions and Software Architecture

During the software architecture phase, the system under development is split into high-level components that communicate using well-defined communication protocols. [11] identifies several software architecture styles, the most popular ones being: *Client / Server*, *Layered Architecture*, *Pipes & Filters*, *Event-Based Architecture* and *Blackboard Architecture*. Since these architectures use different communication paradigms, exceptions within each architecture are of different nature. The following paragraphs discuss the nature of exceptions within each of the above software architecture styles.

Client / Server. In a *client / server* architecture, a client component sends requests to a server component, which executes the request and sends a reply back to the client. This is exactly the situation covered by the idealized fault-tolerant component. In this style there are 3 possible situations in which the server signals an exception to the client: 1) The client's request is malformed or does not conform to the server's specified protocol. In this case, the server signals the appropriate *interface exception*. 2) The request of the client could only partially be completed. The server signals the partial service to the client by means of a *degraded service exception*. 3) The server failed in an uncontrolled way with no guarantees on its internal state. The server signals a *failure exception* to the client. Any handling of one of the above exceptions has to be done in the client. Rigorous exception handling requires that a client must provide a handler for each potential exception. In case the exception can not be addressed successfully by the client, the appropriate exception should be propagated.

Layered Architecture. In a *layered architecture* style, the software is partitioned into several nested layers, where an inner layer provides services to the outer layer. Similar to the client / server style, we propose to use the idealized fault-tolerant component ideas within this architecture style. Rigorous exception handling requires that each outer layer handles all potential exceptions signaled by the inner layer. In case the exception can not be addressed successfully by the layer, the appropriate exception should be propagated.

Pipes & Filters. In a *pipes & filters architecture*, the system is partitioned into several filter components. Each filter reads from its input port a stream of data, processes it, and produces a stream of data on its output port. Some filters also have a control port that can be used to change the way the filters processes the data. Filter components are connected to each other using pipes. Simple pipes connect the output of one filter to the input of another filter. Complex pipes can connect multiple filters by merging or splitting data flows.

In addition to a standard output port, some filters also have an error port. In exceptional situations, the filter component can output data to the error port. Hence, an exception in the pipes & filters architecture is represented by data flowing through an error output port. A pipe can be used to connect an error output port to the input or control port of some other filter. That filter (and the following filters, if any) can then handle the exception.

Event-Based Architecture. In an *event-based architecture*, the individual components are not statically connected. Instead, a set of event types are declared that components can instantiate, in which case all components that expressed interest for that event type receive a notification. Communication is hence anonymous, i.e. the receivers of an event do not know who sent it.

In the event-based architecture style, exceptions are represented by event types that are only instantiated to signal the occurrence of an exceptional situation that needs to be addressed. Any component that is registered to be notified of the exceptional event can provide handling functionality.

Blackboard Architecture. In a *blackboard architecture*, communication between components is centered around a data repository – the blackboard. Components observe the data posted on the blackboard, and as soon as they find data that they can process, they remove the data from the board, process it, and put the result back on the board.

In a blackboard architecture, an exceptional situation manifests when the data on the blackboard satisfies a certain

exceptional condition. Typically, a handler component repeatedly scans the data on the blackboard, and, if it determines that the exceptional condition is satisfied, it executes its handling functionality.

In order to accommodate the different architecture styles, the definition of exceptions at the software architecture phase has to be fairly general:

> *Proposition* 3: An exception at the *software architecture phase* (at a level of abstraction where the software is partitioned into coarse-grained components that communicate using different communication stlyes) is any *exceptional output* E_c produced by a software component to signal to other components the occurrence of an exceptional situation. The nature of the exception depends on the concrete architecture chosen:
>
> - *Client / Server*: An exceptional reply is sent from the server to the client when a service request is malformed, or when the request was only partially completed or failed.
> - *Layered Architecture*: An exceptional reply is sent from an inner layer to the enclosing layer when a service request is malformed, or when the request was only partially completed or failed.
> - *Pipes & Filter*: Data is output on an error port.
> - *Event-Based Architecture*: An exceptional event is sent.
> - *Blackboard Architecture*: The data on the blackboard satisfies an exceptional condition.
>
> Any component that reacts to the exceptional output is a "handler" component.

If exception handling is applied rigorously at the software architecture phase, then the components become error-containment regions, i.e. modules from which an erroneous state can not spread unnoticed to other parts of the application. Filho et al. show in [3] how to extend an architecture description language (ADL) with exception flow information. Their approach proposes to map the enhanced architecture descriptions to Alloy [5], a formal modeling language and tool. This allows the developer to automatically validate the conformance of the architecture to specified exception handling and propagation rules.

3.4 Exceptions and Design

During the design phase, a solution that implements the functionality of each individual component is designed. In an object-oriented design (or any other design that provides modules with interfaces and encapsulation capabilities), the functionality of the system is implemented by a set of objects that send each other messages. Typically, responsibilities are assigned to objects, and each object then encapsulates the state and behavior related to that responsibility.

Just like in the client / server architecture, each object can be designed according to the concepts introduced by the idealized fault-tolerant component. It should raise an interface exception if the service request is malformed, and return an exception to signal a partial or failed request execution.

> *Proposition 4*: An exception at the design phase (at a level of abstraction where each software component has to provide its functionality using a set of interacting modules) is an *exceptional response* E_{op} propagated out of a module to signal the fact that an operation could not be completed as requested. An exception could occur in the following situations:
>
> - The caller module's request is malformed or does not conform to the callee module's specified protocol. In this case, the callee signals the appropriate *interface exception*.
> - The request of the caller module could only partially be completed. The callee module signals the partial service to the caller by means of a *degraded service exception*.
> - The callee module failed in an uncontrolled way with no guarantees on its internal state. The callee signals a *failure exception* to the caller.

3.5 Exceptions and Implementation

During the implementation phase, the constraints of the implementation platform have to be taken into account, e.g. constraints imposed by the operating system or the hardware. Modern programming languages usually provide a set of predefined exceptions to signal violations of platform constraints, e.g. *ReadOnlyDevice* or *OutOfMemory* exceptions, to the program. They are signaled by the language run-time support, i.e. the libraries that encapsulate operating system services.

> *Proposition* 5: An exception at the implementation phase (at a level of abstraction where each operation provided by a software design module is composed of a set of executable statements) is an *exception* E_{st} signaled by the operating system or language run-time when an implementation-related exceptional situation prevents the execution of a statement.

4. MAPPING BETWEEN EXCEPTIONS AT DIFFERENT PHASES

The previous section has established that exceptions are represented in different ways within each software development phase. Although the nature of exceptions changes depending on the level of abstraction that a developer is working on, there exist interesting relationships between the exceptions of each phase. Fig. 3 illustrates these relationships.

The most obvious mapping between exceptions is top-down. Exceptions and handlers (depicted E and H in Fig.3) at a high level of abstraction usually encompass many exceptions and handlers at lower abstraction levels. This is illustrated in Fig. 3 by means of arrows pointing downwards.

A context-affecting exceptional situation $E_s(ca)$ is mapped to one or several exceptional messages $E_m(ca)$ sent by exceptional actors to the system in order to notify it of the context change. Likewise, an exceptional service defined as a handler $H_s(ca)$ maps to a handler operation $H_m(ca)$ triggered by the exceptional message(s). However, if the service consists in multiple interactions between the environment and the system, *only the first* input-output interaction sequence

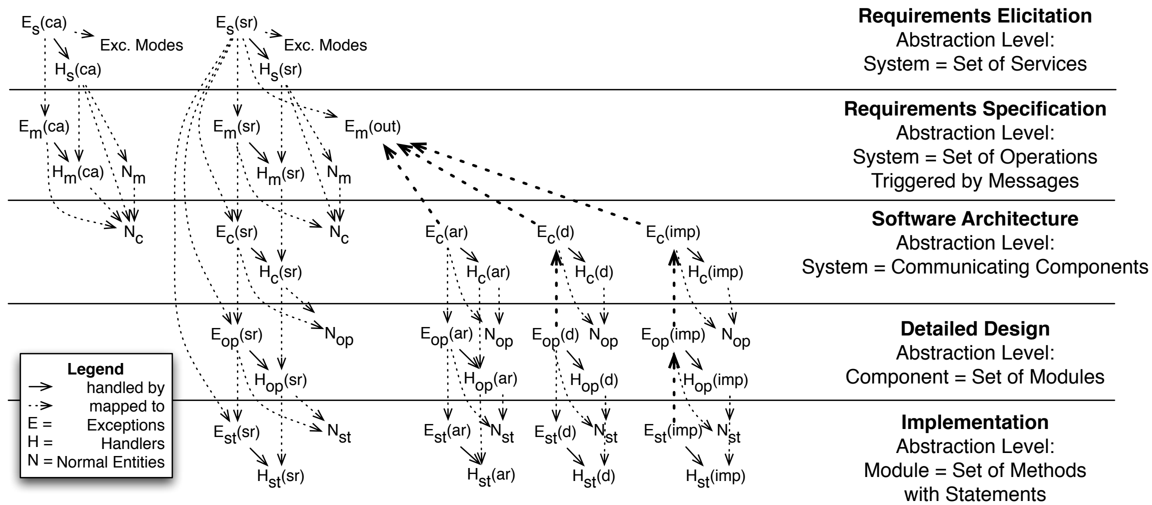

Figure 3: Mapping Between Exceptions at Different Development Phases

is actually a handler operation triggered by an exceptional message. The following interaction steps of the exceptional service are triggered by "normal" messages and provided by standard operations (depicted N in Fig. 3). For example, in case of a fire outbreak in the elevator, the fire detector (exceptional actor) sends the *FireDetected* exceptional message to the system. The system handles that message by instructing the motor of the elevator cabin to move towards the ground floor. Subsequently, the ground floor sensor sends a "normal" message to the system when it detects that the cabin is arriving at the ground floor. This triggers the execution of normal system behavior, during which the system instructs the motor to stop.

A service-related exceptional situation $E_s(sr)$ is mapped to zero or more exceptional messages $E_m(sr)$ (zero if the exception is represented by the absence or abnormal sequencing of normal messages). For each service-related exceptional message, a corresponding handler $H_m(sr)$ describes the first interaction steps involved in handling the exception. Subsequent interactions sequences, if any, are most likely triggered by normal messages. If a requested service cannot be provided, then the system should raise a corresponding exceptional output message $E_m(out)$.

The mapping of exceptional messages to exceptional communications between components is less straightforward. If a component is responsible for communication with a secondary actor, then it communicates the failure of the actor to the other components using exceptions. Therefore, *some* (but not all) service-related exceptional messages $E_m(sr)$ are related to exceptional communications $E_c(sr)$ at the architecture level.

It is also possible that *new* architecture-related exceptions $E_c(ar)$ have to be introduced into the system because of the chosen software architecture. For instance, in a distributed client / server system, it is possible that a server crashes and cannot be reached anymore. Client components can of course define handlers $H_c(ar)$ for such exceptions that try to achieve the desired behavior in a different way. For instance, a crashed server can be tolerated by sending an identical request to a backup server. If the architecture-related exceptions cannot be handled by a component in such a way that the functionality of the failing component can be compensated for, then it is impossible to provide the service, and hence the environment has to be notified of the problem. This requires the addition of new exceptional output messages $E_m(out)$ at the requirements specification level. It is hence possible that exceptions at a lower level of abstraction affect the higher level! This is shown in Fig. 3 by bold arrows pointing upwards.

Similarly to what happens at the component level in a client / server setting, exceptions are used by modules within the design of an individual component in order to signal the failure or exceptional outcome of a method call to the calling module. As a result, some of the exceptional responses $E_{op}(sr)$ are related to a service-related exceptional message $E_m(sr)$ or an exceptional communication among components $E_c(sr)$. New exceptions $E_{op}(d)$ also arise due to the chosen design solutions. If handling *cannot* be done using handler operations $H_{op}(d)$ within the component, then the exception has to be propagated to other components by means of an exceptional (design-related) communication $E_c(d)$. Finally, if the problem cannot be handled transparently at the architecture level, the problem has to be signaled to the environment with a corresponding exceptional output message $E_m(out)$.

Some service-related exceptions $E_m(sr)$ map even down to an implementation-level exception $E_{st}(sr)$. For instance, a message that is sent to a secondary credit card company actor can raise a *CommunicationException* when executing the statement that tries to contact the company server. Therefore, new statement exceptions $E_{st}(imp)$ related to specific implementation decisions have to be defined, together with handlers that address the situation. For example, an application executing on a computer might encounter an *OutOfMemory* exception when trying to allocate a big data structure on an execution platform with memory constraints. Again, if the problem cannot be dealt with within the design module, the appropriate design-level exceptional response should be signaled to the caller module, which might result in new architectural exceptional communications $E_c(imp)$, and in rare occasions in new exceptional output messages $E_m(out)$, if transparent handling of the

implementation platform problem is not possible.

5. POTENTIAL BENEFITS

Integrating exceptions into the entire software development life cycle promises many benefits.

Separation of Concerns. The obvious benefit is clear separation of concerns. Handlers make it possible to separate exceptional behavior from normal behavior, and exceptions unambiguously define the conditions in which normal processing is interrupted and exceptional processing begins. Separation of concerns not only improves readability of the software development artifacts, but also facilitates other software development activities, e.g. debugging.

Prioritizing. Separating the handlers from the normal behavior allows the developer to define special policies and priorities for anything that addresses exceptional situations. For example, handlers at the requirements elicitation and specification phase can be classified into handlers that ensure system *safety* or handlers that increase system *reliability* [9]. If this classification is established, the developer can set clear guidelines that specify that, for instance, safety is more important than reliability. Such a decision affects the software development activity as well as run-time execution of the software. The project manager could, for instance, decide to spend more development and testing time on safety-critical components. At run-time, safety-related exceptions and handlers can execute with higher priority, i.e. they can interrupt even reliability-related activities.

Traceability. The previous section has highlighted the relationships among exceptions and handlers at different software development phases / levels of abstraction. Documenting these relationships during the development process provides valuable traceability information. It allows the developer to propagate high-level decisions down to the lower levels of abstraction, or, conversely, to understand the global exceptional execution context when working on software entities at lower levels of abstraction.

6. CONCLUSION

This paper has investigated the relationship between exceptions and the different phases of software development. We argued that exceptions are of different nature depending on the level of abstraction that the system under development is looked at.

At the highest level of abstraction, where the system is just composed of a set of provided services, exceptions are exceptional situations that prevent the system from providing normal service. At the requirements specification level, where the system interface is the main focus and the system itself still has no internal structure, exceptions are exceptional messages that flow between the environment and the system. At the software architecture phase, where the system is partitioned into a set of communicating components, exceptions take the form of any means of exceptional communication among components. During detailed design, where each component is designed as a set of communicating modules, exceptions are exceptional response messages sent to signal the failure of a service provided by a module. Finally, at the implementation level, where the behavior of each module is implemented using a set of programming statements, exceptions are pre-defined exceptions raised by libraries or the language run-time.

We have outlined a mapping between the exceptions at the different levels of abstraction. The mapping is consistent with, but far more complex than the one presented in [1]. We also showed that some exceptions at a low level of abstraction can require the definition of corresponding exceptions at a higher level of abstraction in the case where transparent handling at the low level is not possible.

Finally, we outlined the potential benefits of integrating exception handling into the entire software development life cycle. If these benefits are indeed observable when applied to a real software development project remains to be investigated.

7. ACKNOWLEDGMENTS

This research has been partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC).

8. REFERENCES

[1] R. de Lemos and A. Romanovsky. Exception handling in the software lifecycle. *International Journal of Computer Systems Science and Engineering*, 16(2):167 – 181, March 2001.

[2] C. Dony. Exception handling and object-oriented programming: Towards a synthesis. In N. Meyrowitz, editor, *4th European Conference on Object–Oriented Programming (ECOOP '90)*, ACM SIGPLAN Notices. ACM Press.

[3] F. C. Filho, P. H. da S. Brito, and C. M. F. Rubira. Specification of exception flow in software architectures. *Journal of Systems and Software*, 79(10):1397–1418, 2006.

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

[5] D. Jackson. Alloy: A lightweight object modeling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, April 2002.

[6] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison–Wesley, 1992.

[7] J. L. Knudsen. Better exception-handling in block-structured systems. *IEEE Software*, 4(3):40 – 49, May 1987.

[8] P. A. Lee and T. Anderson. Fault tolerance - principles and practice. In *Dependable Computing and Fault-Tolerant Systems*. Springer Verlag, 2nd edition, 1990.

[9] S. Mustafiz, X. Sun, J. Kienzle, and H. Vangheluwe. Model-Driven Requirements Assessment for Dependable Systems. *Software and Systems Modeling*, March 2008.

[10] C. M. F. Rubira, R. de Lemos, G. R. M. Ferreira, and F. C. Filho. Exception handling in the development of dependable component-based systems. *Software Practice and Experience*, 35(3):195–236, 2005.

[11] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.