NTNU
Innovation and Creativity

An Examination of Issues with Exception Handling Mechanisms

Christian Tellefsen

Master of Science in Informatics
Submission date: June 2007
Supervisor: Tor Stålhane, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem description

Exception handling mechanisms are an important part in achieving robust and reliable software.

The project will look into what makes exception handling mechanisms difficult to use. Next, the project will focus on methods for achieving better control over exception handling.

Abstract

Exception handling suffers from a fluke in its evolution. Some time in the 1970's, a researcher called Goodenough introduced the exception handling mechanism, known today as the `try...catch` construct.
At about the same time, two fellows called Parnas and Würges published a paper about "undesired events". This paper appears forgotten. The funny thing is, Parnas and Würges effectively described how to *use* exception handling mechanisms.
There is a need to respond to this undesired event. Exception handling suffers from lack of design guidelines and a number of inconsistencies with the object-oriented paradigm, among other things. The thesis' main contribution is creating a library of exception challenges and the evaluation of safety facades, an approach that introduce an architecture and guidelines for designing exception handling. Through qualitative evaluation, this thesis shows how safety facades form an interesting new approach to exception handling.

Preface

This thesis is the result of implementing **the Feynman Problem Solving Algorithm**:

1. Write down the problem
2. Think very hard
3. Write down the answer

Incidentally, the algorithm proved insufficient. A big thanks to my supervisor, Dr. Tor Stålhane, for his suggestions and help. Although I haven't been his most frequent visitor, each visit left me both inspired and informed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

In modern programming languages like C++, C# or Java, error handling is likely in the form of exception handling. Error handling has always been difficult. Exception handling is a step forward from return codes, global error values and gotos, but introduce new problems. So many problems in fact, that in 1982 a doctoral dissertation[8] was written arguing that exception handling constructs

- are more difficult to read
- permits non-local transfers of control, reintroducing the dangers of the goto and the problem of clearing up
- creates a rift between a mathematical function and a software function, and makes reasoning about programs more difficult
- may be used to deliver information to a level of abstraction where it ought not be available, violating the principle of information hiding,
- If a list of possible exceptions is made part of a procedure's interface, that interface is thereby complicated, and interfaces are the very part of a program which one wishes to keep as small and simple as possible. On the other hand, if exceptions are not so specified then the dangers of using them are increased, and many of the advantages of strong typing are lost.

The opinion of Black [8] was overly pessimistic in my opinion. While I disagree with his final conclusions, the problems he listed are still highly relevant.

While it is widely accepted that exception handling has a number of problems, it is the best we currently have available[38, 72].

Exception handling constructs contains three elements:

- The protected block, containing the code that may raise an exception.
- The catch clause, that determines what exceptions should be caught.
- The recovery code, or exception handler.

Example 1.1.1 (The `try...catch` construct)**.**

```
try {
        // This may raise an exception
        doSomething();
        doSomethingElse();
} catch (SomeException e) {
        // Triggered if SomeException is raised.
}
```

In this thesis I will examine why exception handling is so hard to get right, focusing on object-oriented software systems. I will compile a library of challenges with exception handling.

Having established that, I will examine a number of solutions that have been proposed to make exception handling more effective. In particular, I will evaluate safety facades.

Safety facades[66] organizes a system into risk communities. Each community is wrapped by a safety facade that handles all exception handling. The safety facade is replaced depending on the needs of the risk community users. A composition manager may be responsible for picking the correct safety facade, e.g. one designed for batch mode vs. interactive mode.

The article describing safety facades is light on details, and while the idea appears interesting, it is still in an early stage of development.

My thesis will show that safety facades is a viable framework for systems with exception handling. This assertion will be backed up through

- existing knowledge from similar solutions,
- theoretical effects on a number of identified exception handling problems, and
- implementation experiments.

1.2 Prestudy: Exception handling in practice

During the autumn of 2006, I did a minor survey to see how the software industry uses exceptions. I sent out e-mails to nine professionals[1], counting four "senior developers", three "junior developers", one "system architect" and one "requirements engineer". The e-mail, written in Norwegian, asked the following questions:

> Do you have written guidelines on the use of programming language features?
>
> Do you have corresponding guidelines for how your software products should be designed, and for the software development process?
>
> In what degree does these guidelines mention error handling and exception handling? Are there any requirements on how exceptions should be used or designed? Are these of the type "avoid catch(Exception e)", or do they include requirements for architecture/design?

[1] All recipients were aquaintances.

All nine answered my e-mails. Five respondents answered they did not have exception handling guidelines for their systems. Four of these had no guidelines at all, except for standard Java programming guidelines. One answered that exception handling guidelines were used in one of the projects he worked on. The remaining three respondends answered the following:

Large software company, business software: Guidelines contained in a language independent document describing programming guidelines and code conventions. An internal Wiki expands on these guidelines.

Large software company, communications software: Only the java programming guidelines are used, but an exception handling framework has been explicitly designed into one of the current projects. Extensive rules for the development process and software design, but no guidelines for exception handling.

Consultant, large firm: "Exception handling is very important", states that exception handling is handled early in their software projects.

The mini-survey also showed that smaller companies have fewer guidelines

than larger companies. Only three respondents had an offensive approach to exception handling.

Only one respondent claimed to have well documented guidelines that also focused on exception handling. Unfortunately these guidelines were considered business secrets and could not be disclosed.

1.3 Research methodology

This thesis consists of three stages of research:

1. A chart of challenges for exception handling.
2. A chart of possible solutions to the challenges.
3. Evaluation of the Safety Facade architecture.

Stage one and two will in large parts be the result of an extensive survey of the literature. The information gained from these surveys will provide background knowledge for stage three.

Stage three will be a qualitative evaluation of the safety facades. This evaluation poses a bit of a problem. The original description of the safety facades lacks detail, and appears to be at an early stage of development. To my knowledge, there are no reference solutions or other systems with safety facades available for study. Developing some example implementation appears to be neccessary. The safety facade evaluation will consist of the following parts:

- A qualitative evaluation on how the safety facades affect known exception handling challenges.
- A qualitative evaluation of safety facades vs. software compartments, a similar, more tested design approach.

- Java example code, to uncover pitfalls and provide reference code.

The purpose of this research is not to find detailed information on the effects that follows by using safety facades in a real environment. At the current stage of development, I believe safety facades benefits from experiments and theoretical examination. Having taken care of common pitfalls, and created example solutions and guidelines, safety facades will be better prepared for further experimentation.

Research method evolution

At an early stage, it became clear that while safety facades were highly interesting, there was minimal information available, no reference solutions, and no known implementations. Although I later found that a third party J2EE library ha implemented support for a version of safety facades, I have found no software that have used this implementation. In addition, I found a number of research papers describing how exception handling affects software development, but no single paper gave "the complete view".

As a result, I decided to begin by collecting the background information now in the chapters *Challenges* and *Proposed Solutions*. During this work, I found the proposed design approach called safety facades, and decided to examine it. The overall objective did not change much, my goal has been to find a way to make exception handling easier to use.

Informatics research methods

Being a product of human behaviour, software engineering tools and methodologies must be empirically tested[69]. Theoretical discussions and experiments are useful only up to a certain point.

Depending on the research questions, it seems most relevant to use lab experiments or a case studies to evaluate safety facades empirically.

A lab experiment is by nature limited in scope, but allows greater control than both case studies and action research. A lab experiment involving safety facades could involve either refactoring an existing system of limited size, or design a system from scratch. Implementing a system from scratch may be too time consuming and would introduce additional variables.

A case study would cover a larger problem than a lab experiment. The best situation would be to observe a real software development project, in part or entirety. If the group of test subjects have worked on similar projects earlier, those may prove useful as a baseline.

The products, or software, could be evaluated for

- measurable benefits, quantitatively or qualitatively,
- how the system's structure is affected, qualitatively

Focusing on the human aspect, possible topics are

- what problems the test subjects encountered,
- effectiveness, or did safety facades affect the number of work hours
- how the test subjects interpreted the safety facade information

The main point here is to do the testing in an environment that is as real as possible, and that the research method is mostly dependent on the research questions. There are two objects that may be researched, the construction process and the final product.

Chapter 2

Background

2.1 Terminology

This section will introduce basic terminology related to exception handling. I will start with basic quality attributes, continue with general terminology, and conclude with definitions related to exceptions in object-oriented software.

Terms such as *error*, *exception* and *reliability* are ambiguous due to different use within the software engineering community. Other terms have subtle differences, like error vs fault.

Terms like error and reliability implies measuring a system's observed behaviour against its desired behaviour. In practice, measuring actual behaviour against a specification.

Definition 2.1.1. specification. A document that specifies, in a complete, precise, verifiable manner, the requirements, design, behaviour, or other characteristics of a system or component, and, often, the procedures for determining whether these provisions have been satisfied. [10, p. 69]

There are two problems with specifications. First, specifications are hard to get right[19]. Incorrect or insufficient specifications lead to software with unwanted behaviour, despite conforming to the specification. Second, specifications are not written to include all possible system inputs and states. Because they are hard to discover, exceptional situations are less likely to be included.

To create a solid point of reference for discussing desired and undesired behaviour of a system, some simplifications are neccessary. I assume specifications to be correct and comprehensive. This paper does not discuss the issue of understanding and implementing a specification correctly.

The lack of specification for undefined exceptional behaviour will be discussed later.

2.1.1 Quality Attributes

Reliability and robustness is closely related to error handling. While several definitions exist[62, p. 11], the following definitions are sufficient for my purposes:

Definition 2.1.2 (reliability)**.** The ability of a system or component to perform its required functions under stated conditions for a specified period of time[10, p. 62].

This implies that reliability concerns whether a system is able to perform its tasks as required by a specification. Robustness on the other hand, focuses on what would happen if the system were subject to abnormal situations:

Definition 2.1.3 (robustness)**.** The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions. [10, p. 64]

Robustness may be considered a subattribute of reliability. While robustness concerns the possibility that unexpected input may crash a system, reliability concerns the ability of a system to function as desired. Exception handling is closely related to both.

2.1.2 Error, Fault and Failure

There are subtle differences between the words *error*, *fault* and *failure*. These words are also used differently depending on context. Some definitions are in order.

Definition 2.1.4. An *error* is the symptom or manifestation of a fault in software. The error may be observed as an invalid measurement, result or unanticipated behaviour.

Example 2.1.1. A method $\mathcal{M}$ in object $\mathcal{A}$ is reading from a file. While reading, invalid input is detected. An exception is propagated[1] to the caller of $\mathcal{M}$. Reading is aborted, but the file resource is not freed. The error is observable through resource leakage.

While errors are measured directly, their source is often difficult to identify. Note that this paper does not consider hardware faults.

Definition 2.1.5. A *fault* is the underlying cause for an error, e.g. erronenous code or a design problem.

Example 2.1.2. Recall the above example. The fault exists in the error handling code of method $\mathcal{M}$, which does not close the file resource.

Having defined a fault and the measurable error, we need to be able to discuss the system-wide implications of these.

Definition 2.1.6. A *failure* is the inability of a system to deliver services as stated by the specification[10, p. 32]. A failure may either be *soft* or *hard*. A soft failure implies a degraded state, with partial functionality available. A hard failure "results in complete shutdown of a system"[10, p. 36].

Example 2.1.3. The system in the former example will experience a failure when file resources are depleted. If the system crashes or becomes inoperable, it suffers a *hard failure*.

[1] Defined in section 2.3

It is only relevant to discuss *failure* of a system in relation to a given *specification*. As stated in 2.1, this paper assumes specifications to be correct.

Having defined error, now consider *error handling code*:

Definition 2.1.7 (Error handling code)**.** The parts of a software system responsible for detecting and handling abnormal situations in the system or in its environment.

Error handling is a part of any non-trivial system. Note the difference between error tolerance (the system does not fail upon input errors) and *fault tolerance* (the system does not fail, despite the presence of software faults[2].)

2.2 Error handling

Traditional error handling in a procedural language typically consists of calling a function and checking its return value (or other known side effects).

Example 2.2.1 (Error handling in C with return values)**.**

```
int foo() {
        if (do_work() == -1) {
                // Handle errors from do_work()
        } else if (more_work() == -1) {
                // Handle errors from more_work()
        } else if (...) {
                ...
        }
}
```

A value may also be returned through a function parameter passed by reference, or using status values like the global errno variable in C and C++.

Example 2.2.2 (Error handling in C with global error variable)**.**

```
int foo() {
        int work_result = 0;
        do_work();
        if(errval) {
                // handle error for do_work()
                // ...
                return -1;
        }
        work_result = more_work();
        if(errval) {
                // handle error for do_workQW
        }
        // do something useful with work_result
        // ...
}
```

Buhr and Mok [16] highlights some of the drawbacks with traditional error handling.

- A caller must explicitly test return values or status flags.

[2]Paraphrased from [10, p. 33]

- These tests will be located throughout the program, reducing readability and maintainability.
- It may be difficult to determine if all error cases are handled.
- Removing, changing or adding return or status values is difficult as the testing is coded inline.

2.3 Exception handling

The origin of software exception handling is probably hardware traps. A hardware trap is used to signal a CPU that a some special event has occurred. Upon a hardware trap, or signal, the operating system triggers a special handler that processes the event. This might be a notice that the hard disk has completed a data request.

Exception handling mechanisms was introduced in the 1960's and spread through focus on software reliability during the mid-1970's[63]. Much of the terminology used today is attributed Goodenough in his 1975 paper "Exception Handling: Issues and a proposed notation"[30].

The word *Exception* have several meanings within the computer science dicipline. It may describe a hardware exception (e.g. divide by zero), an abnormal situation in the software state, or as a programming language object representing an error situation. The latter example may indicate a manifestation of the two former situations.

Buhr and Mok begin their article[16] with "(...) but there is hardly any agreement on what an exception is." Their paper avoids the problem of defining an exception by considering the exception handing process as a whole, and exception is a component of an exception handling mechanism (EHM). An EHM is used for directing control flow after an abnormal situation has been detected.

This text looks into how to best make use of the exception handling mechanism. This text will consider exceptions as normal objects, as in Java, C# and C++.

With the list of problems in the previous section in mind, it is clear that an exception handling mechanism must (quoted from [16])

1. alleviate testing for the occurrence of rare conditions throughout the program, and from explicitly changing the control flow of the program
2. provide a mechanism to prevent an incomplete operation from continuing, and
3. be extensible to allow adding, changing and removing exceptions.

Exception handling mechanisms includes special exception handling constructs to the programming language. Code for normal situations are separated from code handling abnormal situations. This contrasts traditional error handling, where the programming language provides few or no error handling features.

Exception handling adds a second exit option to the interface of subcomponents. A function may return either return normally or raise an exception. That

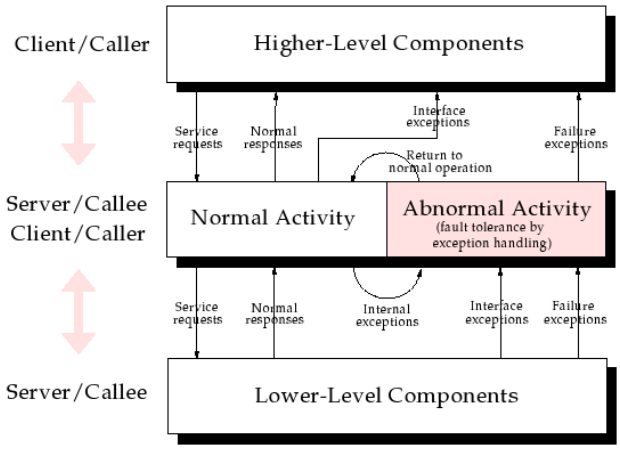

Figure 2.1: Conceptual model of exception handling, from [27]

is, the function may abort its execution and signal its caller that an abnormal situation occurred.

Statements that may throw exceptions may be wrapped in a *protected block*. Each protected block has one or several *handlers* thattriggers on exceptions raised within the block. The handler is tasked with getting the running program back to a normal state. It may also re-raise the exception, or raise a new exception.

Exceptions raised without an enclosing protected block are propagated out of the current subroutine. The propagation continues to higher level components until an appropriate handler is found. If no appropriate handler is found, the program is terminated. During propagation, the stack is unwinded as neccessary.

The languages C++, C# and Java use similar language constructs for exception handling with the `try...catch` construct. Other languages use different syntax, but the main principles are the roughly the same.

In Example 2.3.1, exceptions of type `SomeException` are handled by the associated handler block. Other exceptions are propagated to a higher level. It is also possible to define handlers that trigger on any thrown exception, see Section 3.4.1.

Example 2.3.1 (The `try...catch` construct)**.**

```
try {
        // This may raise an exception
        doSomething();
        doSomethingElse();
} catch (SomeException e) {
        // Triggered if SomeException is raised.
}
```

If `doSomething()` raises an exception, the exception type is compared to `SomeException`. If it matches, control is turned over to the catch-block. If not, the exception is propagated to a higher level component.

Example 2.3.2 (The `try...catch...finally` construct)**.**

```
try {
        // This may raise an exception
        doSomething();
        doSomethingElse();
} catch (SomeException e) {
        // Triggered if SomeException is raised.
} finally {
        // This is always executed.
}
```

The `finally` block is always executed, regardless if an exception was raised or not. To see what happens if an exception is raised within a finally block, see Section 3.4.3.

When the catch-block is completed, control is turned over to the first statement following the entire `try..catch` block. That is, control does not return to the try-block.

The approach above is called the *termination model*. Two other approaches exist, the *resume model* and *retry model*. These are mostly abandoned today, after being susceptible to programming errors. In addition, both models may be simulated using the termination model. E.g., the retry model may be simulated by combining a loop with a `try..catch` block.

In the `try...catch` example, the exception is *internal* if it is handled within the current stack frame. If the exception escapes the current component (method, procedure, function), it is an *external exception*, and becomes part of the components interface. A programming language may require external exceptions to be explicitly added to the interface, called *checked exceptions*. This enables the compiler to check for unhandled exceptions during compilation.

2.3.1 Special Exceptions

Some exceptions should not be caught. In Java, they are called Errors, and are thrown by the JVM, signalling critical failures like stack overflow, memory depletion or coding errors. In C#, similar semantics are carried by, among others, the StackOverflowException and ExecutionEngineException. These exceptions cannot be caught by a normal catch clause in C#. Errors may be caught in Java, but it is highly discouraged.

2.3.2 Object Oriented exception handling

C# and Java are highly object-oriented. The exceptions are complete objects, and may contain fields and methods like any other object. In C++, there are fewer restrictions for exceptions, although there are guidelines. In other languages, exceptions may be defined as some type of signal, a primitive data type, or as a special symbol. That is, exceptions may have different properties than other user-defined data types.

In both C# and Java, all exceptions have a single supertype. In Java, this is the class `Throwable`. Checked exceptions are subtypes of `Exception` (subtype of Throwable), unchecked exceptions are suptypes of `RuntimeException` (itself derived from Exception). Abnormal situations signalled by the Java Virtual Machine derive from the `Error` class (subtype of Throwable). C# has a similar but different hierarchy, with only unchecked exceptions.

From this hierarchy, a large tree of exceptions are supplied with the Java system. Programmers may add their own exceptions to the tree.

When exceptions are objects, they may contain methods and fields. This makes it possible for an exception to provide a wide array of information to its handlers.

Subsumption is relevant for exceptions as well. A catch-clause of the type $\mathcal{E}$ may trigger when an exception of type $\mathcal{E}$ itself is raised, and when a subtype of $\mathcal{E}$ is raised. Accordingly, a catch clause for `Throwable` in Java, will trigger on any raised exception.

Example 2.3.3 (Example of subsumption)**.**

```
class AppException extends Exception { ... }

class AppConfigException extends AppException { ... }

// ...

try {
        doSomething(); // Throws AppConfigException
} catch (ApplicationException ae) {
        // Triggers on ApplicationConfigException by subsumption
        // Restore state, handle problem ...
}
```

2.3.3 Interface exceptions

Exceptions are separated into two groups, *internal exceptions* and *external exceptions*. Internal exceptions are handled within the subroutine it was raised. External exceptions are propagated outside the subroutine it was raised.

However, unlike normal return values, exceptions do not neccessarily need to be declared in the interface. Programming languages may require all or some exceptions to be declared, or may not require declarations at all. Exceptions that must be declared are called *checked exceptions*, and those who do not are called *unchecked exceptions*. When exceptions are explicitly specified, they may be called *interface exceptions*. This is ambiguous though, since both checked and unchecked exceptions are part of the interface.

The words *checked* and *unchecked* are related to the compiler's type checking. The compiler may easily ensure that a checked exception is either handled or propagated. Unchecked exceptions are not subject to the compiler's control. Data-flow analysis (DFA) may be used for analyzing the flow of unchecked exceptions, but the analysis is imprecise.

Methods that throw checked exceptions, must explicitly declare these in the method interface. The exception type may not be enough information: Under what circumstances can the exception be raised? What state will the component be in when the exception is raised?

When working with proprietary libraries and COTS software, the source code may be unavailable. Without thoroughly documented interfaces, the exceptional behaviour will be unknown. Even when source code is available (e.g. OSS), the code may still be too vast to investigate properly.

Stroustrup [68, p. 936] recognizes the problem of object states when throwing exceptions. First consider the two following guarantees:

- No guarantees: If an exception is thrown, any container being manipulated is possibly corrupted.
- Strong guarantee: If an exception is thrown, any container being manipulated remains in the state in which it was before the standard-library operation started.

Both are unacceptable, the latter because rollback is costly and might be impossible. Stroustrup designed the objects in his C++ Standard Library (STL) to conform to one of the following *exception-safety guarantees*:

- "*Basic guarantee* for all operations:" The basic invariants of the standard library are maintained, and no resources (...) are leaked.
- "*Strong guarantee* for key operations:" In addition to the basic guarantee, the operation succeeds or has no effects (...)
- "*Nothrow guarantee* for some operations: In addition to providing the basic guarantee, some operations are guaranteed not to throw an exception."

Generally, the minimal requirement for any method is the basic guarantee. In addition, it should be clear from the exception context what state the originating object has after raising the exception, and if possible how the exceptional situation might be resolved. For instance, a collection of objects are to be manipulated, but the operation fails half-way. The exception context should describe how far the operation got before crashing. It will then be possible for the caller to retry the operation on the remaining objects, revert the changes or perform good logging.

2.3.4 Checked vs unchecked exceptions

The choice of checked vs. unchecked exceptions has major implications for modifiability and cohesion. Unfortunately, there are no clear lines separating the types.

Authorative sources on Java[51] suggest the following division of responsibilities:

Checked: Exceptions that signal an exceptional situation which might be treatable.

Unchecked: Exceptions that signal an untreatable situation.

This raises the question: What is treatable, and what is not? A method raises exceptions it does not know how to handle. How does the method know if an exception is "treatable" at all? The article contains a detailed list of criteria for each type. One of these are that all projects should define an unchecked `ProgrammingException` that is the root for programming mistakes, e.g. `NullPointerException`. Although one of the clearest and thorough pieces of documentation on the use of checked vs. unchecked exceptions, there are still room for mistakes.

Proponents of *Design by Contract* divide the responsibilities like this:

Checked: Errors that are part of the contract.

Unchecked: If the contract (assumptions) are broken.

This can be shown to give almost the same division as the guidelines above. Here, the division is determined by a methods contract. Programming errors breach the contract and results in unchecked exceptions. Business level exceptions (e.g. withdrawMoney() throws NotEnoughMoneyException) are defined in the contract, and implemented as checked exceptions.

Another issue is deciding when to convert a checked exception to an unchecked one and vice versa. This problem appears when exceptions are abstracted during propagation. A FileNotFoundException is checked, but a missing configuration file would indicate an incorrect installation. Here, the exception would be converted into the unchecked InstallationException.

Experts disagree whether checked exceptions are good or bad[49, 65, 70, 71, 72]. The only major modern programming language using checked exceptions is Java, while unchecked exceptions are found in several languages, including C++, C# and Python. Some research has been done on introducing both checked exceptions and contracts to C#[38].

In short, proponents of checked exceptions claim they add important information to method interfaces and that the drawbacks are avoidable through good guidelines and standards. Opponents of checked exceptions claim they are only syntactic sugar and greatly reduce modifiability. In my opinion, the biggest problem with checked exceptions is deciding when to use them. This is solved by standardized guidelines and better education of programmers.

2.3.5 Runtime performance

Exception handling has implications for runtime performance, both before and after an exception is thrown. Exceptions are, ideally, rarely thrown, so any performance loss should be postponed until an exception pops up.

During normal execution, information about where to find exception handlers is maintained. One approach is adding appropriate pointers to the stack frame. When an exception is thrown, the exception object is created. Next, a search for the closest fitting handler is performed. The trigger clause of each handler in the active stack frame must be examined. If no handler is found, the stack is unwinded and the search continues. When propagating the exception, finally blocks may also need to be executed.

The point is, throwing an exception is resource intensive, while adding try...catch blocks are not.

In real-time systems and other systems where runtime performance is important, the overhead of throwing exceptions is a problem[37]. One solution is to sacrifice initialization costs in favor of more predictable resource use when throwing exception, i.e. make it easier to find handlers.

Another solution is to minimize the number of exceptions that are thrown. This will also increase the predictability of a program, according to Amey and Chapman [5]. [68, p. 937] introduce the "Nothrow guarantee" for exception safety.

In the next part, I will look further into the challenges of exception handling.

2.4 Software Architecture

What is software architecture? From Bass et al. [7] we have the following definition:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of these elements, and the relationships among them.

Ideally, a system's architecture is a set of elements that interact using well-documented interfaces. The less ideal end of the scale is the well-known *big ball of mud* with *spaghetti code*.

Architecture is the abstract definition of a system. Although the elements of an architecture might be implemented as objects, they need not be. The architecture shows *structure and purpose*, while the implementation defines *how* the system works.

If single objects are not really of interest, then what role has exceptions in the software architecture? Quite an important one, it seems. This is illustrated in the interface documentation template suggested by Bass et al. [7, p. 212]:

> **4.** *Exception definitions* These describe exceptions that can be raised by the resources on the interface. (...) Common exception handling behaviour can also be defined here.

Bass et al. [7] documents the flow of exceptions in the component-and-connector view for fault tolerance. This is quite interesting. Specific error situations shows up as early as during architectural design. It should not be too surprising though, since it is already known that exceptions may have global implications. It is prudent to get an overview of error situations at an early stage of development.

Chapter 3

Challenges

What makes exception handling difficult? In this chapter, I will present several of the contributing factors to exception problems. I will begin with general issues, like complexity, readability and availability of proper tools. Following this introduction, object-oriented design, implementation and architecture level problems will be presented.

3.1 General issues

Getting a grip on the exceptional control flow is hard. It is difficult to find all possible error situations and exceptions that a method might raise. Getting error handling right is well known to be difficult. Perhaps the foremost challenge with exception handling, is that any exception may have global implications in the software. To the individual software developer and architect, exception handling requires different solutions than traditional error handling.

Any statement may possibly raise an exception. With traditional error handling, it is known that a function may report an error a return code, in a parameter, or in a global error variable. Exceptions on the other hand, may be raised from any statement, and they may be handled at a completely different place than where the function is returned.

It has even been suggested that exception handling makes it harder to distinguish between good and bad code[20]. With traditional error handling, it is easy to see if the return values are handled or not. Not neccessarily so with exceptions.

During a code review experiment[6], 42 participants were asked to identify problems with the code, including exception handling. The code was altered to contain bugs. One of these was an empty catch clause, which only 7 participants identified as an error. The paper points out that exception handling was the last point on the checklist [1], which may have resulted in "the fatigue effect".

[1] Software inspections are typically carried out with a checklist with what to look for. In this case, the two last questions were:

- Are all relevant exceptions caught?
- Is the appropriate action taken for each catch block?

Programmers must also learn new idioms, patterns and techniques to make best use of the new language constructions. Considering this, it is interesting to note the near complete lack of focus on the subject in education and literature.

Most introductory software engineering textbooks appear to marginalize or ignore error handling. A few mention that error handling is important, or mention techniques from the fault tolerance dicipline, like n-version programming. Exception handling design is almost completely missing, except in a few books (like Budgen [14, p. 200], Meyer [46] and Braude [12]

Modelling languages, like UML, also lack support for exceptions. The latest UML version 2.0 (2004-2005) added support for modelling exceptions in activity and sequence diagrams. Some researchers argue for even better support [32, 57].

3.1.1 Analysis tools

Techniques like data flow analysis becomes less accurate because branching from exceptions is added[67]. More complex algorithms are required to get good analysis results. Source code analysis is used by, among other things, code optimalization techniques in compilers, test-coverage computations and by visualization tools.

Software may be analyzed *statically* or *dynamically*. Static analysis allows examination without actually running the software, and includes control-flow analysis, data-flow analysis and control-dependence analysis. Dynamic analysis is performed by running the software, and is used by debuggers and profilers. Static analysis is faster and easier to perform, but gives approximated results. Dynamic analysis is slower, but may collect more accurate data.

Control-flow analysis

Control-flow analysis results in a control-flow graph, and determines "for each program statement s, those statements in the program that could follow s in some execution of the program"[67]. Control-flow analysis is essential for several other techniques, including data-flow analysis and control-dependence analysis. . Exception handling enables almost any statement to branch off to an exception handler. The analysis must also consider the catch- and finally-blocks.

Data-flow analysis

Data-flow analysis is a generic term for "techniques that compute data flow facts, such as definition-use pairs, reaching definitions, available expressions, and live variables"[67] at various stages of the program. Compilers use several data-flow techniques when performing various optimizations, like subexpression elimination and copy propagation[3].

Data-flow techniques must be updated to handle two new situations. First, exception propagation may cause different code to be executed, adding new definition-use pairs. Second, use of the exception variables needs to be included in the definition-use pairs. The latter requires connecting where exceptions are thrown with where they may be caught.

Control-dependence analysis

From Sinha and Harrold [67], "control-dependence analysis determines, for each program statement, the predicates that control the execution of that statement." It is used for program slicing (used by debuggers, regression testing, etc).

Most algorithms for control-dependence analysis performs *intraprocedural* analysis. However, finding all statements that may trigger a catch clause would require *interprocedural* analysis.

Tools

Analysis and testing of exception handling constructs requires well-implemented algorithms to be able to reason about the exception flow. The problem is that exception handling constructs may cause arbitrary changes in control flow, both within procedures (intraprocedural) and across procedures (interprocedural). There is a some literature describing how exception handling affects software analysis and testing[13, 23, 24, 26, 42, 59, 61, 67, 73, 74], but good tools have yet to emerge from this research. Most tools are proof of concepts and lack the quality required for common use[52].

3.2 Object-Oriented design

Exception handling may conflict with many of the goals of object-oriented design (OOD). Miller and Tripathi [49] studied how exception handling and object-oriented design work together. In their work, four major areas of conflict were introduced:

- Complete exception specification
- Partial states
- Exception conformance
- Exception propagation

3.2.1 Complete exception specifications

Complete exception specifications is an interesting notion. Most programming languages only include exception *types* in object interfaces — if exceptions are mentioned at all. The result is interfaces with insufficient information describing exception flow. Four extensions are suggested, to provide complete exception specifications. Quoted from [49]:

> - Exception masking: how is the abstract state of an object affected if the implementation masks an exception.
> - Exception consistency: each exception that can be signaled has an associated meaning that is consistent regardless of the signaling location.
> - Exception context: what is the exception context provided for each signaled exception

- Object state: the state of the object immediately prior to the exception signaling is indicated within the exception context.

The two last points needs further explanation. *Exception context* is in a later article, the authors explain that "An exception context corresponds to an execution phase or region of a program."[48]

Object state is *normal* upon normal exists. An exception occurrence may need to put the object in an invalid state of some sort. The object may require instructions from another component to choose recovery action. This should be a part of a complete interface specification.

3.2.2 Partial states

Consider a composition of objects with a defined structure, for instance a directed acyclic graph. A composition is in a partial state when each object in the composition has a valid state, but some relationships between objects are invalid.

When objects are added or removed, invariants for the linked list must be maintained. If an exception occurs when adding or removing objects, the composition may be left in a partial state.

Individual objects may be left in a partial state, when a method operating on its data is aborted by an exception. This leaves the objects internal pointers in an undefined, probably invalid state.

3.2.3 Exception conformance

With object-oriented inheritance, a parent object $\mathcal{P}$ may be used in place of a more specialized child object $\mathcal{C}$. That is, object $\mathcal{C}$ is conformant with $\mathcal{P}$ because it implements (as a minimum) the same interface as $\mathcal{P}$. Miller and Tripathi [48] introduce *exception conformance* for the special case when objects $\mathcal{P}$ and $\mathcal{C}$ are exception types. Exception conformance is useful, but may conflict with normal object-oriented ideas. From Miller and Tripathi [48]:

> The difference between exception handling and normal object-orientation is that methods can be overloaded to have similar meaning in a wide variety of situations, but exception information (particularly for errors) generally needs to be specific.
>
> (...) There is a conflict between exception conformance and complete exception specification. To have a complete specification implies exception conformance, but to allow evolutionary program development suggests exception non-conformance, which in turns suggests an incomplete exception specification.

Although Miller and Tripathi has a point, it is not always relevant. Exceptions may be useful, even when they are not specific. The usefulness of information contained by an exception is not immidiately determined by how specific an exception is defined. An IOException may provide enough information in many situations, and still make room for "evolutionary program development." An IOException becomes insufficiently specific only when the low-level error is neccessary to choose the appropriate recovery action.

3.2.4 Exception propagation

Exceptions may change the program control flow. Suppose method $\mathcal{P}$ raises an exception to its caller $\mathcal{Q}$. Method $\mathcal{Q}$ regains control, even if $\mathcal{P}$ has not completed all its instructions. Normally, $\mathcal{Q}$ would expect that $\mathcal{P}$ had completed, but now the programmer must consider how to handle an exceptional return from $\mathcal{P}$.

3.2.5 Object-Oriented goals

How does exception handling affect the central object-oriented concepts *abstraction*, *encapsulation*, *modularity* and *inheritance*?

Abstraction

Miller and Tripathi [48] divides *abstraction* into *generalization of operations* and *composition*. Exception handling supports generalization of operations by enabling callers to decide what should happen in case of abnormal situations. The downside is that exceptions may break the abstraction by disclosing internal information.

The second form of abstraction is composition. A composition may consist of a composition object $\mathcal{C}$, which is an interface to the composition, and objects arranged in a structure, e.g. a linked list or a tree where some invariants must be maintained.

An object in a composition may raise an exception — but how should the exception be propagated? Standard EHMs only allows an exception to propagate upwards the call-chain. It may be neccessary to notify the composition object $\mathcal{C}$, or one or more members of the composition. If an external caller has direct access to the objects in the composition, the exception would not be registered by the composition object $\mathcal{C}$. Additionally, indivitual objects in the composition may not know which other objects to notify.

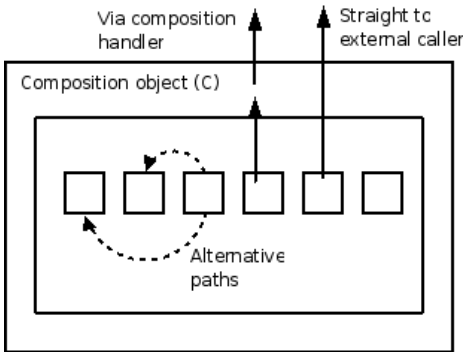

Figure 3.1: Conceptual view of a composition where exceptions are thrown from within the composition. The dotted lines illustrate exception propagations not supported by EHMs.

Encapsulation

Encapsulation hides the internal structure of an object, and presents its users with an interface containing a subset of the objects resources. If clients of the object use ”inside information” to circumvent the interface, the encapsulation is broken and the interface is in practice extended to include the inside information. This will have a major impact on modifiability, as well as modularity.

Exceptions may violate the encapsulation through disclosing

- conceptual information, e.g. how the object is constructed,
- from where the exception is signalled, or
- pointers to internal data.

It may be impossible to completely maintain the encapsulation of an object when using exceptions. Exceptions must by nature disclose some internal information to describe what has gone wrong.

Modularity

Modularity is accomplished when a change in one software component (or module) does not require changes in another component. Miller and Tripathi [48] states three exception related problems related to evolution through incremental changes:

- Exception Evolution: When new specialized exceptions are created as subclasses of existing exceptions. Old handlers may be triggered by these exceptions which may not handle the exceptional occurrence correctly.
- Function Evolution: When the functionality of a method is changed, ”(...) unchanged modules may have to cope with new exceptions that are introduced by the new functionality.”[48]
- Mechanism evolution: When the implementation of a method is changed, the existing exception types may become overloaded with new semantics. They may signal new situations or require different handling.

Checked exceptions should also be mentioned here. Recall that all checked exceptions thrown by a method must be declared in the method interface. Adding or changing checked exceptions may require updating a large number of methods. Opponents of checked exceptions claim the increased dependencies far outweighs the advantages[72]. Proponents claim checked exceptions are a useful tool and does not negatively affect modifiability if used properly[71].

Inheritance

There are also two problems related to *inheritance*. First, a subclass may attempt to specialize error handling from one of its parents methods. If the original exceptions do not contain enough information for specialization, the subclass may need to reimplement the entire parent method to throw specialized exceptions.

Second, suppose a parent class calls a virtual method that is reimplemented by a subclass. The parent may require new handling code for exceptions thrown by the new method. This would require reimplementing the parent method.

3.3 Exceptions and Architecture

Software architecture literature describes exception handling in context of the fault-tolerance dicipline. The main focus is how the architecture can be designed to minimize the impact on faulty components.

Current architectural description languages delegates actual error handling to the lower level components. Research is being done to examine how one can include more of this functionality in the architecture itself[28, 29, 34].

The following issues are somewhat related to the implementation level, but their implications are on an architectural level.

3.3.1 Anonymous Exceptions

Aanonymous exceptions are often described in the ADA literature. If an exception is propagated outside its scope, it becomes anonymous. Where the anonymous exception occurred will be unknown, and it can only be caught by `when other`[2].

A similar problem exists in other languages. If an exception is propagated outside its scope, its type may be unknown and it cannot be explicitly caught. Fortunately, unlike in ADA it may still be possible to extract an error message and stack trace.

3.3.2 COTS and external components

Anonymous exceptions are especially interesting when externally written components are included in the software, i.e. COTS and OSS, or simply legacy code.

Closed or unknown source code will throw unexpected exceptions. Interface documentation for exceptions is typically sparse. The exceptions a method may throw is either unknown or limited to a list of checked exceptions.

3.3.3 Exception abstractions

Abstraction is important, also for exceptions. When propagating an exception, an exception's semantics should always conform to the surrounding code. Failure to create a proper hierarchy of exceptions risk creating a spaghetti of exceptions flowing around in the system.

Example 3.3.1. An SQLException might be abstracted as DatabaseException or DataStoreException. Higher layers probably does not know anything about SQL, why should they receive an *SQLException*?

Abstraction has the additional advantage of reducing the number of exceptions flowing across the architectural elements. For instance, Robillard and

[2]Equivalent to `catch(Throwable t) { ... }` in Java

Murphy [60] shows how they reduced the number of exceptions in *Jex* from 70 to 14 through sound design of the exception hierarchy.

3.4 Implementation level

Programmers make errors for a number of reasons, from simple typing errors and poor use of programming language features, to complex errors made from lack of overview and invalid assumptions. This part will examine mistakes that are known to happen regularlyBundy and Mularz [17], Howell and Vecellio [33], McCune [45], Müller and Simmons [51].

Some of these examples are elsewhere described as anti-patterns, the opposite of software design patterns like Singleton, Observer, Visitor, etc. Anti-patterns are bad solutions to specific problems and demonstrate typical pitfalls.

3.4.1 Plain abuse

These mistakes are either directly harmful, like the empty catch clauses, or are used in a confusing way, like using exception handling for normal control flow.

Normal control flow

Exceptions are reserved for abnormal situations. For normal control flow, standard langugage mechanisms should be used. A classic example is using an ArrayIndexOutOfRangeException[3] to indicate the end of an array during iteration.

Example 3.4.1.

```
int[] numbers = {1, 2, 3, 4, 5, 6}; // Initialize array with integers
int sum = 0;
int i = 0;
try {
        while(true) {
                sum += numbers[i]
                i++;
        }
} catch (ArrayIndexException aie) {
        // Reached end of array                                    10
}
```

Raising an exception is resource intensive: An exception object must be created, a handler must be found and the handler must be set up correctly. Using this mechanism as a part of normal execution is inefficent and complex.

Using exceptions for normal control flow may also cause confusion whether a signaled exception is an error condition or a normal situation. In the example above, the ArrayIndexOutOfRangeException could have signaled an actual error when accessing the array. That situation would be masked as *we are finished*.

Early papers did consider using exception handling for normal control flow[30].

[3] Array index out of range

Null handlers

Null handlers are an easy way to avoid handling an exception, or postpone the decision of how an exception should be handled[33]. This is a special case of *Invalid Propagation Termination*, see Section 3.4.3.

A checked exception must either be caught or or declared in the interface of the methods that may raise it. In the example below, the programmer expects the configuration file to be present at all times. By silently swallowing the exception, the code is not cluttered with `throws` sentences and lets the programmer avoid or postpone writing a handler.

The FileNotFoundException would be suppressed, but the error would immidiately resurface when attempts are made to read from the FileReader.

Example 3.4.2.

```
try {
        FileReader fr = new FileReader("myprog.conf");
} catch(FileNotFoundException e) { }
```

Usually, this exception would have to be propagated up the call chain until a gui-module or other component could display an error message (or automatically regenerate the configuration file with default values).

A stopgap measure is to rethrow an unchecked exception. This ensures the exception is noticed, while still postponing writing a good handler, and prevents the system from continuing in a corrupted state[51].

A single exception for everything

Application specific exceptions add great value to the existing hierarchy of builtin exceptions. The new exceptions enables transfer of exception information as well as maintaining information hiding. However, rethrowing all exceptions as a single application specific exception is a design bug.

Example 3.4.3.

```
public void myMethod() throws ApplicationBaseException {
        try {
        ...
        } catch (IOexception ie) {
                throw new ApplicationBaseException(e.getMessage());
        }
        ...
}
```

Müller and Simmons [51] argues this reduce the usefulness of checked exceptions in Java. In effect, all exceptions would become either checked or unchecked, and the `throw` clauses would give little information to the caller.

The catch-all clause

In Java, all exceptions derive from the superclass Throwable, while unchecked exceptions derive from RuntimeException. In C++, most standard library exceptions derive from exception. A *catch-all clause* is a catch clause for any exception, as demonstrated in Example 3.4.4. The C++ equivalent is catch(...) {}.

Example 3.4.4 (Catch-all in Java)**.**

```
try {
        // do something
} catch (Exception e) {
        // Recover.
}
```

There are several dangers with this approach:[31]

- The handler will be triggered for *all* exceptions, including the unexpected ones.
- No exceptions are propagated. If your code does not handle a specific exception, it may be masked (or swallowed, disappear). This is difficult to detect during testing.
- Even though all exceptions are currently handled properly, later modifications may introduce new exceptions that require different handling.
- Debuggers may be thrown off track. Exceptions are caught by the catch-clause, even if they are not actually handled.

In some exceptional situations (pun intended), the catch-all construct may be appropriate. It should only be used after considering the side effects. One such use is a last resort handler, to perform logging, rescue user data or displaying an error message before crashing.

A stop-gap measure to fix some of these issues is demonstrated in Example 3.4.5. By immidiately rethrowing the exception, the stack trace is not lost, while some cleanup is possible. This might be a possible app

Example 3.4.5 (Less dangerous catch-all)**.**

```
try {
        // do something
} catch (Exception e) {
        // Recovery code.

        throw; // Rethrows e
}
```

In practice, most catch-clauses should look like Example 3.4.6

Example 3.4.6 (Better rewrite)**.**

```
try {
        // do something
} catch (FileNotFoundException e) {
        // Recovery code.

        // ...
        // Signal to callee if neccessary,
        // by return value rethrowing the
        // exception or throwing a
        // new exception.                                    10
}
```

Asserts in catch clauses

Assertions are typically used to verify the state of internal data and pre- and postconditions of functions during development and testing.

Example 3.4.7 (Simple assert() example.)**.**

```
#include <assert.h>
// ...

float divide(int dividend, int divisor) {
        assert(dividend != 0);

        return dividend/divisor;
}
```

If dividend is zero, the program will crash immidiately, rather than crash or fail at some arbitrary place. In Java, a failed assert will throw the unchecked exception AssertionException.

Assertions are usually disabled in production systems. Assertions may be enabled at the flick of a switch (e.g. Java) or the system may need to be recompiled (e.g. C++ software). Shore [64] argues assertions is an excellent tool and should always be turned on.

Mixing assertions and exceptions may introduce pitfalls. If an assert were used in a catch clause, it would result in an empty catch clause in the production system.

Asserts are not a replacement for exceptions. In Java, unchecked exceptions may be used.

3.4.2 Information loss

Exceptions are a great source of information for debugging. During exception propagation, a stack trace should be assembled, and logging performed where appropriate. Many languages, including Java and C#, create a stack trace automatically.

Stack trace

When an exception is propagated, its propagation path is recorded in a stack trace. When exceptions are rethrown, the stack trace must be preserved. Java 1.4 added support for *exception chaining*.

Example 3.4.8 (Exception Chaining)**.**

```
void read() throws ServerUnavailableException {
        ...
        try {
                ...
        } catch(SocketException se) {
                // Only get message, lose stack trace.
                throw new ServerUnavailableException(se.getMessage());
        }
}
```

10

```
void read() throws ServerUnavailableException {
        ...
        try {
```

```
            ...
        } catch(SocketException se) {
            // Wrap old exception, retain stack trace
            throw new ServerUnavailableException(se.getMessage(), se);
        }
}
```

The former example will lose the trace while the latter will add the stack trace to the new exception.

Incomplete logs

In context of Java, Müller and Simmons argues exception logs should contain the most information possible to facilitate debugging:

> We see no need for restraint here, since the cause for the problem could potentially be anywhere in the system, and a resolution of the problem has highest priority.

Exceptions should only be logged once, but as much information as possible should be included. Some issues in regard to logging are

- individual instances of exceptions logged several times
- insufficient information (e.g. ”Module X crashed”)
- missing state information
- incorrect information
- irrelevant information

Java programs has the advantage that the JVM rarely crashes, which ensures a correct stack trace. C++ programs are prone to crashing while propagating exceptions. This raises a number of implementation issues. When designing a system for logging, robustness is a primary requirement. The logging subsystem must not be affected by errors in other parts of the system.

3.4.3 Other issues

Invalid Propagation Termination

Empty catch clauses, or null handlers, were introduced above. Another variant is shown in the example below. Here, a stack trace is printed, but no attempt is done to return the system to a valid state. The problem is that the user is unlikely to see the stack trace. Even if he did, he would not recognize it as a problem, thinking ”Excellent, it didn’t crash”, and continue to use the system.

Example 3.4.9.

```
try {
        ...
} catch (AppException e) {
        e.printStackTrace();
}
```

Writing exception handlers requires thorough knowledge of the operation that failed, as well as the surrounding system. To properly handle an exception, the system must be returned to a valid state. Some of the corrective actions that the handler must consider are:

- release resources (e.g. memory, file pointers)
- maintain invariants of methods and collections
- revert an incomplete operation
- retry an operation
- safely terminate the program

Improper or missing release of resources is a regular problem. In a recent study of "mistakes that involve resource leaks and failures to restore program-specific invariants", Weimer [73] analyzed 4 MLOC, finding almost 1200 bugs.

In the example below, an exception raised at line 6, would leave rs still open. Moving the two close statements to the finally block is not a solution. In case of an exception at line 5, the statements `rs.close()` and `ps.close()` will fail because rs and ps are uninitialized.

Example 3.4.10.

```
Connection cn; PreparedStatement ps; ResultSet rs;
try {
        cn = ConnectionFactory.getConnection(/* ... */);
        StringBuffer qry = ...; // do some work
        ps = cn.prepareStatement(qry.toString());
        rs = ps.executeQuery();
        ... // do I/O-related work with rs
        rs.close(),
        ps.close();
} finally {                                                    10
        try { cn.close(); } catch (Exception e1) { }
}
```

The example is adapted from Ohioedge CRM[4], by Weimer [73]. Some form of incremental control is neccessary. Resources must be released in the proper order, and only released if actually acquired. Two solutionsWeimer [73, p. 25] are,

- nested try-finally blocks.
- sentinel values or run-time checks, i.e. only run `rs.close()` if rs is not null and opened.

Suggestions have been made for language enhancements that allows automation of this bookkeeping. More on this in Section 4.4.

[4] An open-source CRM system.

Checked and unchecked exceptions

Java is the only widely used language with both checked and unchecked exceptions. Other languages, like C++ and C#, has only unchecked exceptions.

Using only checked or unchecked exceptions should be considered a design flaw in Java.

Introducing too many *checked* exceptions, will result in code littered with `throws` clauses. Changeability and maintainability would suffer, since every propagated exception would need to be specified at every method interface along the propagation path.

By only using *unchecked* exceptions, you get no assistance from the compiler to identify unhandled exceptions. You also lose exception information that would otherwise be included in the method interface.

Inconsistent use of the two types of exceptions is difficult to avoid without clear guidelines for their use. See Section 2.3.4.

Exception semantics

An exception should have the same meaning no matter where it is thrown. It may abstract a number of situations, but these should be clearly defined and closely related.

Example 3.4.11. A `NullPointerException` in Java should never be used to signal an empty stack. According to its specification, this exception class is thrown when null is supplied where an object is required. Similarly, an `IOException` signals a number of different error conditions, but they are all I/O-related.

Exceptions that are not part of the built-in framework must also carry equal semantics regardless of where they are thrown. Three remedies are

- unambiguous naming schemes
- well documented exceptions
- a well-designed exception hierarchy

Exception semantics must also be balanced so they are not too general in order to make proper use of catch clauses. If you regularly need to run tests on an exception to deduce the actual error, the exception may be too general.

Uncaught exceptions

If an exception is thrown, but no valid handler is found, your program will crash. This is undesirable, although better than masking it with an empty catch clause. Doing so may leave your program running in an invalid state (Section 3.4.3).

During development and maintenance, new exceptions may surface ”all over the place”. Handlers for these may be overlooked, removed or misplaced. Detecting these defects are not trivial, since you need to know how each exception may be propagated. Data-flow analysis may help, but the information it produces is not accurate.

Disclosure of implementation details

An exception signals that something wrong happened in a component. To make recovery or logging possible, some information must be passed along the exception. Any information may be passed, from an integer to a large data structure.

Example 3.4.12. The Java ArrayIndexOutOfBoundsException signals an attempt to access an invalid array index. Its "payload" is simply the invalid index number.

Although a bit contrived, if propagated outside of the current class or module, this exception would expose how the internal data structure is implemented (an array).

The disclosure of implementation details is a violation of *information hiding* and *abstraction* principles, but is often unavoidable when using exceptions [49]. The need to convey information must be balanced with the neccessity of information hiding. Consider also that the throwing method has no control over how far an exception might propagate.

In the example above, it would be better to throw something like a RangeException or the built-in IndexOutOfBoundsException. The former would hide the fact that the internal implementation uses a numbered scheme for accessing elements.

Disclosure of internal data

While Section 3.4.3 concerns hiding *structure and concepts*, now consider how internal *data* might be exposed.

Exceptions contain fields and methods like other objects. The fields are used to convey the exception context. This information should not include pointers to internal data in the throwing object, as it would break the information hiding and add unwanted dependencies.

Insufficient information for handling

An exception must carry enough information for correct handling. The information should be readily available. Sometimes, the exception type itself is enough information.

Example 3.4.13. The NullPointerException conveys enough information just by being raised. This exception is thrown by the JVM when an attempt is made to dereference a null pointer. The stack trace in combination with the exception type will pin-point the place and type of the error.

An error during an HTTP connection requires signaling both that an error has occurred, as well as the HTTP status code. The HTTPException stores the HTTP status code and makes it available to its handlers.

The type of information is important as well. Error codes, file names, indexes, state of the operation and other types of information contained in the exception context should be conveyed as an appropriate data type. Many exceptions simply provide a human readable error message. This is unfortunate, because it is prone to changes, hard to standardize (even Oxford Press does not standardize English) and often leads to string parsing hacks.

Example 3.4.14. The HTTPException solves this problem by providing the error code via getStatusCode() as well as a human readable error message through the toString() method.

Unfortunately, even the Java API contains many exceptions that only use a readable string. Bloch has little good to say about parsing exception messages:

> In the absence of such methods, programmers have been known to parse the string representation of an exception to ferret out additional information. This is extremely bad practice. Classes seldom specify the details of their string representations; thus string representations may differ from implementation to implementation and release to release. Therefore code that parses the string representation of an exception is likely to be nonportable and fragile. [9, p. 173]

Mapping from return codes to exceptions

Howell and Vecellio [33] found many bugs when converting between different error handling paradigms. When implementing a Java-application that use a C-library, it may be useful to perform a conversion perform a conversion fromm error codes to exceptions. The problem however, is that programmers tend to forget that return codes may change.

When return codes (and error codes like errno) are converted, it is important to convert all existing error codes, as well as safeguard against new error codes. In the example below, the `default` block makes sure that any unexpected error codes are not lost.

Example 3.4.15.

```
class NativeC {
        public static native read_data();
        static { System.loadLibrary("NativeC"); }
}

class Example {
        public void doWork() throws DataFileException, DataValueException {
                int retcode = NativeC.read_data();

                switch retcode {                                                    10
                        case 0: // ok
                                break;
                        case 1: // File not found
                                throw new DataFileException();
                                break;
                        case 2: // Invalid data
                                throw new DataValueException();
                                break;
                        default: // Handle unexpected situations
                                throw new ProgrammingException("Unexpected return code " 20
                                        + retcode + " from NativeC.read_data()");
                }

        }
}
```

Propagation from within handlers

Almost every line of code may raise an exception, including statements within `catch` and `finally` blocks. Programmers tend to forget this small fact, according to Howell and Vecellio [33].

The finally block in C# and Java, shown in the example below, is especially important to look into. The purpose of the finally block, is to guarantee that a particular bit of code is always executed, even in the presence of exceptions.

Now, suppose the ProgrammingException was thrown, and the finally block is executed. What would happen if the "cleanup code" were to raise an exception?

Example 3.4.16.

```
try {
        // do something
} catch (IOException e) {
        // Rethrow as new exception
        throw new ProgrammingException("message", e);
} finally {
        // Cleanup code.
        // Always executed, even if exception is thrown from try/catch
}
```

The original exception will be discarded and replaced with the exception thrown from within the finally block, which is most likely not what you would want. Where appropriate, it should be considered to add a nested try..catch. This is possible both in the catch- and finally-blocks. Avoid propagating checked exceptions from the finally block.

Chapter 4

Proposed Solutions

The previous chapter highlighted a number of exception handling issues that plague software development. In this section, I will look at possible solutions to many of these problems. The solutions are mostly found in scientific journals and conference proceedings. Some tools are available for practicioners, like the Fault Simulator and Ballista, introduced in Section 4.2 below.

Exception handling should be managed through the entire development process. This requires control of exception usage in both the development process as well as individual phases. As early as requirements engineering it is possible to gather information on how the software may fail, possible errors from the environment.

It is neccessary to standardise how exceptions are defined, designed and used in the system. Choosing techniques for elaborating exceptional requirements, design, and testing promotes consistent exception handling use. As I have shown earlier, it is difficult to manage exception flow and behaviour. Interfaces often contain insufficient exception information for proper handling. This makes it especially beneficial to promote uniform behaviour.

4.1 Exception identification

Knowledge of exceptional or abnormal situations that a system could experience or cause is important for two reasons. First, this information is key for correct handling of abnormal situations. Thus, abnormal situations should be found during requirements engineering. The second situation is testing, where it is also neccessary to know what abnormal situations there are, and how they should be handled by the system.

Unfortunately, imagining every possible error situation a system might encounter is notoriously difficult. To understand why, Maxion and Olszewski [43] points to cognitive science and the study of human errors.

Human errors may be classified into two categories. Doing something the wrong way is a commission error, while forgetting to do something is an omission error. A review of nuclear power plant incidents(Maxion and Olszewski [43], quoting Reason [58]) showed that near 70% of omission errors are related planning, testing, modification and recall, tasks that are similar to designing software and charting exceptions.

Checklists are a simple way to prevent omission errors, but they are more useful for concrete tasks than "find all possible error scenarios". When they are too long, they become difficult to remember. Also, checklists tends to make their users less imaginative.

Another technique is mnemonics. The CHILDREN mnemonic[43] lists different faults that a system might encounter. The author suggests using this mnemonic to initiate a fishbone diagram, often used for brainstorming.

C	omputational problem
H	ardware problem
I	/O and file problems
L	ibrary function problem
D	ata input problem
R	eturn-value problem: func or proc call
E	xternal user/client problem
N	ull pointer and memory problems

Table 4.1: The CHILDREN mnemonic

Some techniques have been adapted for software from other engineering diciplines. These are group activities that all use some formal procedures to encourage thinking about error situations. I will briefly present a few of these:

- Failure Mode and Effects Analysis (FMEA)
- Fault Tree Analysis (FTA),
- HAZard and OPerability analysis (HAZOP), and
- Safety cases

Failure Mode and Effects Analysis (FMEA)[35, p. 433], were developed by German rocket scientists in the 1940's. The software adaptation use a subset of the modern FMEA. FMEA is a step-by-step process that begins with defining the system and reliability requirements, before determining how its components may fail and the estimated rate of failure. This is used to estimate how the failures effect the entire system, and results in suggestions for design improvements. Software FMEA is used to evaluate both early and later design, and gives estimates on the software's properties. Highly simplified, FMEA attempts to answer the following, for a system's components:

- Failure mode: "In what ways could this subsystem or component fail?".
- Effects of failure: What effects would the failure have, as perceived by the customer?

Fault Tree Analysis[35, p. 440] were also a result of rocket science, this time from the Bell Laboratories' work on the Minuteman ICBMs in the 1960's. Fault trees answer how a failure is triggered. A tree is constructed for each failure, e.g. Web page not displayed, which is the root node. Using logic gates like AND and OR, the basic events that lead up to the failure may be graphed. A web page may fail to appear because of the basic events "Network unavailable" and the web page has either never been visited, or has been expunged from a local cache.

Hazard and operability analysis (HAZOP) were originally developed for the chemical processing industry. It has since been adapted for a number for other applications, and research is being done on using it in software engineering[18]. HAZOP is a group activity, where guide words are used to encourage thinking around how the system handles abnormal situations. If the guide word *less* is used while examining a network connection, one may come to realize that the system should be tested with degraded network speed or high packet loss. Both general as well as software engineering or project guide words may be used.

A safety case is a set of arguments to support the claim that a component hold a certain property. This provides a structure on which to organize safety claims.[18, 43].

A variation on safety cases is the *dependency case*[43]. A dependency case provides an argument that a component is guarded against certain exceptions (or hazards). Maxion and Olszewski [43] suggests using a HAZOP process in combination with the CHILDREN mnemonic to find abnormal situations.

4.2 Testing and analysis

Analysing exception flow in software requires good knowledge of how the exception flow affects the system. Sinha and Harrold [67] describes in depth how control-flow analysis and control-dependence analysis is affected by exception handling mechanisms. To illustrate, consider that a `try..catch..finally` may be entered in four ways and the try, catch and finally blocks have five, four and two possible branches respectively. An algorithm for calculating control dependencies in software with exceptions is also presented. Allen and Horwitz [4] contains a similar model.

The two papers acknowledge that their static analysis techniques still suffer from inaccuracies — false positives as well as false negatives. False positives and false negatives are typical problems for static analysis algorithms.

Robillard and Murphy [61] presents a model that "encapsulates the minimal concepts necessary" for a developer to reason about exception handling in a programming language. The model is useful for researchers, and appears to have been used during development of the analysis tool JeX. JeX uses static analysis techniques to generate a visualization of exception flow.

Nguyen and Sveen [52] examined JeX as well as analyzer plugins for Teamstudio and IBM Websphere Studio. The tools are claimed to be "insufficient". JeX is still experimental and is described by Nguyen and Sveen [52] (in 2003) as outdated. The IDE plugin for Teamstudio contains little exception handling support, but is customizable. Of these three, the WebSphere studio plugin appears to be the most promising tool — it detects a number of antipatterns and exception handling errors using control-flow and data-flow analysis.

Fu and Ryder [26] tracks exception *objects* rather than exception *types* with their *DataReach* analysis algorithm. This reduced false positives considerably, though execution time increased around tenfold.

Some newer tools are the Fault Simulator from Compuware[1] and Microsoft's FxCop[31, 36]. Two experimental plugins[26, 41] for the Eclipse IDE has also

[1] http://www.compuware.com/products/devpartner/fault-simulator.htm

been made. The former is a lint-clone that performs static analysis of the software and produces a list of warnings. The latter adds visualization and various IDE enhancements to Eclipse. Microsoft has hinted they are going to add exception analysis support for Visual Studio .NET[36].

The tools above use control-flow and data-flow analysis or simpler techniques to identify problems in code. The theoretical grounding is sound, but false negatives and false positives are still a problem for these algorithms.

4.3 Dynamic analysis tools

Dynamic analysis performs tests or measurements on running software. Prime examples are profilers, debuggers and test suites. Some approaches are:

- Monitoring during normal execution (e.g. debuggers)
- Black-box fault-injection
- White-box fault-injection

Debuggers are excellent for collecting information and modifying a a running system. Another approach is to instrument code with loggers, like the well-known `println()`. This can be automated to make a program log whenever an exception is thrown or propagated. By hooking into the compiler, instrumentation is added when the program is compiled, completely automating this task. Ohe and Chang [53] demonstrates the value of such logging in combination with a gui tool for monitoring.

Ballista performs black box robustness testing of API-level functions. For each data type in the function declarations, a set of values is defined. A function with a single integer parameter would be called a number of times, each time with a different input value, typically 0, -1, lowest int, highest int, etc. A robustness failure occurs when a parameter or combination of parameters cause the function to crash.

Pan [54] examined the robustness of POSIX[2] implementations in 15 operating systems. On average, 82% of the robustness failures were caused by the input from a single parameter. Mukherjee and Siewiorek [50] suggests robustness tests should be organized into hierarchies for better reusability and consistency.

Fault Injection by injecting exceptions is a different approach. With this technique, program source code or machine code is altered to throw exceptions at various points. The testing shows how the exceptions are handled, or that they are not handled. Fault injection require the software to be recompiled between each test, and preferably that the software may be automated.

Software Fault Injection is often the only way to accomplish a high test coverage in systems with exception handling. Many exceptions are very hard to trigger in a testing environment, like connection errors or resource depletion.

Compared to dynamic analysis, static analysis require fewer preparations and setup, and is much less resource intensive. Dynamic analysis may be more difficult to use and is slower, but solves different problems than static analysis. In Figure 4.3, static analysis and fault injection techniques are compared.

[2]POSIX is an API standard for operating systems. The purpose is to encourage application portability by providing a set of standard functions.

Fault Injection	Both	Static Analysis
Simulated hw faults	File sys race conditions	Invariant checking
Prod/on-line use	Memory Management	Formal verification
Automated "black-box" testing	Ret.value handling	Model checking
Recovery meth. trig/verification		Code style/method. analysis

Table 4.2: Capabilities of fault injection and static analysis, from [13].

4.4 Programming language features

Several researchers have suggested improved exception handling constructs. The general opinion is that the current mechanisms leave much room for improvement.

4.4.1 Automating management of resources

As established earlier, proper cleanup is a problem in the case of exception occurrences. To help this problem, Weimer [73] suggests *compensation stacks*. These are based on ideas from the database literature, *compensating transactions* and *linear sagas*.

Compensating transactions reverse previously committed transactions. Sagas ensure that a series of actions will either be completed, $a_1a_2a_3a_4a_5$, or rolled back with appropriate compensations, $a_1a_2a_3c_3c_2c_1$. CompensationStack objects store compensations for a series of actions.

When an "action" has completed, its associated cleanup is executed. The entire stack (all compensations) are executed when the stack goes out of scope[3], when an exception is thrown but not handled, or at the programmer's discretion. Interfaces are annotated with compensation requirements, e.g. when opening a file, the compensating `fp.close();` is added to the stack of compensations, ready to be executed when the current scope ends — or when the programmer desires.

Microsoft's .NET provides the IDisposable interface combined with the *using* keyword[21, p. 247]. An object that implements IDisposable may be initialized upon entry to the `using`-block. When exiting the `using`-block, its Dispose() member will be called, and the object is marked for garbage collection. This approach is comparable to compensation stacks, although simpler.

Resource dependencies may be organized in a tree structure. By releasing a single parent node, all its children would be released first. Park and Rice [55] claims their Framework for Unified Resource Management is more flexibla than compensation stacks, partly because of the tree structure. Their solution is implemented without the need for language support.

Unfortunately, I believe this makes their solution less intuitive to use, and require more complicated code than both compensation stacks and the C# solution.

The D programming language allows code to be triggered when exiting a scope[1]. The `scope` statement is used to define the code, which may be triggered on normal exists (`success`), exceptional exits (`failure`) or any exit (`exit`). The last defined `scope`-statement is executed first.

[3]Remember the stacks themselves are implemented as ordinary objects

Aspect-Oriented Programming may be used in a similar fashion.

4.4.2 Contracts

Programming with contracts allows the programmer to explicitly state invariants, that may in turn be validated at run-time and compile-time. This helps to guarantee that objects are left in a valid state even when exceptions are raised.

Contracts are added to object- and method interfaces, and may specify preconditions and postconditions for both input parameters and internal states. Spec#[38] is a variant of C# with support for contracts, including checked exceptions. Checked exceptions implements the ICheckedException interface. A method may be declared like this:

Example 4.4.1.
```
class Connection {
        /*...*/
public string ReadMessage(int bytes)
        requires bytes > 0 ;
        throws SocketClosedException
                ensures unchanged (this ^ Connection) ;
        /*...*/
}
```

The value of the parameter bytes must be larger than zero. If a SocketClosedException is thrown, `ensures unchanged` will guarantee that all fields in the current object that are members of the class Connection are unchanged. This is just a few examples of how contracts may be used.

One objection to contracts should be mentioned. Because the contracts are explicitly stated, modifying the source code may require the contracts to be updated, possibly resulting in a ripple effect that spans many methods in a call chain. On the other hand, contracts may provide valuable information for maintenance programmers.

4.4.3 Checked exceptions for module interfaces

Methods that throw checked exceptions must declare these in their interfaces. Changes to a method that throws a checked exception, may require updating the interfaces of all its callers.

It has been suggested that checked exceptions might be omitted for module-internal function calls. Here, *module* is used loosely. External function calls on the other hand, may originate from code that is written by programmers with less information about the called module. The programmers that know the module, may have easy access to its methods and may have better knowledge of its implementation. Methods with different user requirements allows different amounts of interface information.

Malayeri and Aldrich [41] suggests that only module interfaces should require checked exceptions to be declared. This simplifies the use of checked exceptions inside the module, while retaining the exception declarations for external users. An analysis tool is used to verify that checked exceptions are declared at module interfaces.

4.4.4 Anchored Exception Declarations

Anchored exception declarations is another attempt at reducing the extra work associated with keeping checked exceptions updated. The idea is simple, instead of declaring each method to throw a specified exception, the programmer simply states "this method $\mathcal{A}$ throws the same checked exceptions as $\mathcal{B}$".

Example 4.4.2.
```
void f() throws like g(), like h();
void f(A a) throws like a.g() propagating (E1, E2);
void f() throws like b().g() blocking (E1, E2);
```

In the above example, two declarations are shown from an implementation suggested by van Dooren and Steegmans [70]. The first function throws the same exceptions as `g()` and `h()`. The second function only propagates the exceptions `E1` and `E2` from `a.g()`. The third function throws the same exceptions as `b().g()`, except for `E1` and `E2`.

4.4.5 Bound Exception Handlers

Exceptions are caught by types, not by their source. In some situations it might be useful to be able to catch exceptions by matching both exception type and source type[15]. This may help to make exception handling code easier to read.

Example 4.4.3.
```
try {
        ... logFile.write(); ... dataFile.write(); ... tmpFile.write(); ...
} catch( logFile.FileError) {...}       // Bound
  catch( dataFile.FileError) {...}      // Bound
  catch( FileError) {...}               // Unbound
```

Without bound exceptions, the example above would have required a single `try..catch` block around each `write()`-call. Buhr and Krischer [15] shows that if bound exceptions are desired, they must be implemented as part of the language.

The idea of choosing handlers based on exception context has previously been used in languages like Lisp, Smalltalk and Beta. These implementations are more generic than suggested by Buhr and Krischer [15], where the handler selection algorithm may be replaced with arbitrary code.

Earlier approaches has not updated the exception context during propagation. That is, with *fixed bindings*, the object bound to the exception is not updated. As I have shown earlier, adjusting the exception type to the appropriate abstraction level during propagation is important. The same issue exists with the bound object.

Transient bindings, introduced by Buhr and Krischer [15], allows automatically updating the bound object during propagation. This may be done manually by the programmer, by catching and reraising exceptions, or be integrated into the language. If integrated, the algorithm is simple. When the bound object is not visible in the next stack frame, the it is replaced with the current object. This ensures proper information hiding. Fixed bindings on the other hand, may propagate a responsible object outside of its scope, allowing a handler to circumvent information hiding.

Buhr and Krischer [15] believes bound exceptions are better aligned with object-oriented design, since handling may be based on the originating, or responsible, object. The exact implications on code quality is not examined by the authors.

4.4.6 Rescue Handlers

It has been established that one of the problems with exception handling is that callers may be unaware of all exceptions that a method throws. If a method is changed to throw a new exception, the caller must be updated to handle the new exception, or risk invalid handling of this exception.

Rescue Handlers[47] are provided for situations where an exception is thrown, but no handler is found. If a caller does not know how to handle an exception, the exception handling mechanism will search for a rescue handler in the originating object, allowing some form of error handling. An exception may also have a global rescue handler.

A rescue handler may attempt recovery actions, including a rethrow with a different exception type. Support for the rescue handlers must be added to the programming language.

4.4.7 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) allows code to be triggered on events, like every time a method is called, upon leaving a method, throwing an exception, etc.

One purpose of AOP is removing duplicate code. Logging exception throws becomes very simple with AOP — simply provide the command for logging, and state that this code should be run whenever an exception is thrown. Even better, it is possible to attach different snippets of code to different exception types, and their subtypes.

AOP has been demonstrated for a number of programming languages, including Java. First, the programmer writes AOP-code. One way of using AOP is to write AOP-annotated Java code, and then use a tool to generate standard Java code before compiling.

Example 4.4.4[39] shows how an aspect can be used to add code before and after methods. Here, the set-methods is wrapped with calls to println().

Example 4.4.4. *// Connect the setters aspect to the following methods*

```
crosscut setters(): Point &
        (void set(int x, int y) |
         void setX(int x) |
         void setY(int y))

static advice setters() {
        before {
                System.out.println("Entering a setter");
        }                                                          10
        after {
                System.out.println("Exiting a setter");
        }
}
```

Example 4.4.5[25] demonstrates an aspect that is triggered when an SQLException is thrown by the method Connection.setAutoCommit().

Example 4.4.5.
```
public aspect ConnectionPoolHandler {
    pointcut setManualCommitHandler() :
        call(* Connection.setAutoCommit(..));
    declare soft : SQLException : setManualCommitHandler();
    after(Connection con) throwing (SQLException e) :
            setManualCommitHandler() && target(con) {
        con.close();
    }
}
```

Researchers differ in their views on using AOP for exception handling. Lippert and Lopes [39] claims that AOP provides an excellent way to simplify exception handling. They found that there were mainly three reactions to an exception occurrence: ”log and ignore”, ”set the return to a default value” and ”throw an exception of a different kind”. In an experiment where AOP was introduced to an existing system, 414 exception handlers (2070 LOC) were replaced with 31 catch aspects (200 LOC).

Filho et al. [25] on the other hand, makes a convincing argument against AOP. In their experiments, they found that AOP did not reduce the amount of exception handling code. Additionally, because the aspects, that contain the code, are separated from the normal callchain, it is more difficult to find where an exception is handled. To make use of AOP, the authors believes AOP should be included early in the development process. Adding AOP on top of an existing system appears to have little value.

4.5 Design

It seems that exception flow should be a part of software design methods. Various ideas are floating around in academia, from architecture description languages with support for exception handling, to architectural patterns and design guidelines.

Architecture design languages could be extended with exception handling information. The Architectural Exceptions Reasoning and Analysis (AEREAL) framework[24] attempts to introduce exception handling into an architecture. This separates exception handling on the architectural level from the rest of the system and enables some automated verification of the exception handling. Unfortunately, this approach requires extra work, including a formal, textual description of the architecture. Issarny and Banâtre [34] suggests a similar approach.

Garcia and Rubira [28] use computational reflectivity[4] to separate exception handling from normal behaviour and provide common information and handling. A similar approach is suggested for workflow-driven web applications[11].

Introducing exceptions in the architecture promotes early exception design. The models above would help in this way, but they do not appear ready for widespread use yet. They also require the introduction of special languages

[4] A separate architectural layer monitors meta-information and may alter the system structure and behaviour at runtime.

and tools. Although interesting, the models only discuss tools for introducing exceptions into architecture, not exception handling design.

The following sections will look into approaches for good exception handling design. I will start with an article from 1976 that still has not been outdated.

4.5.1 Undesired Events

In 1976, Parnas and Würges [56] published their paper on designing software to handle *undesired events*. This was the same year as Goodenough published his seminal paper on exception handling constructs in programming languages. However, while the recovery block and termination model were embraced by language designers, the design paper of Parnas and Würges appears bypassed by practitioners. This is unfortunate, because while Goodenough mainly describes the language constructs, Parnas and Würges describes how to use the constructs.

Parnas and Würges [56] discusses how a system should be structured to "[facilitate] the introduction of recovery and diagnostic algorithms." The terminology is different from what is used today. They coin the term *undesired event* (UE) to cover about the same as *exception* or *abnormal situation*. This is to avoid the ambiguities of the term *error*[5].

Normal code should be separated from error handling code. By placing error handling code separately, it is easier to facilitate information hiding and changes later during development. Traps are used to transfer control to the error handling routine when an error is detected. In a modern language, exception handling would be used.

Parnas and Würges [56] advocates complete module interfaces. As an "aid to [undesired event] anticipation", the authors state a list of considerations.

- Limitations on the Values of Parameters
- Capacity Limitations
- Requests for Undefined Information
- Restrictions on the Order of Operations
- Detection of Actions Which are Likely To Be Unintentioned
- Sufficiency: "The conditions above are sufficient to guarantee that, if none of them applies, [the function completes] without violating any module limitations. Further, [if no trap occurs] (...) the value of the function (if any) will not be 'undefined'."[6]
- Priority of Traps: If several errors occurs at once, which error is signalled?
- Size of the "Trap Vector": What context information is made available.
- State After the Trap

[5] Recall Section 2.1.2

[6] Note that the routine may still fail, but the reason would be some internal error rather than invalid input.

- Errors of Mechanism: When the module fails. It may be needed to disclose information hidden by the module's abstraction.

The situation with *errors of mechanism* requires a further explanation. How much information should the caller receive? What information is available at all? If the undesired event is completely unexpected, perhaps caused by a programming error, the actual cause and effects of the error would be hard for a program to find. While it is easy to simply report "failure" and crash the system, it is often possible to report information usually hidden by abstraction. What records were changed? What sectors on the disc have gone bad?

One suggestion from Parnas and Würges is to report a failure classification along with the UE, so the following information is available to the user:

- Did the value of any function change?
- Is it worthwhile to retry?
- Are other functions than the current one affected by the error?
- Is the module in "a state consistent with the specifications", or is it in an unknown or invalid state?

In addition, all undesired events are divided into crashes and incidents. Incidents are expected and were corrected successfully, all other errors are crashes. Now, the important question is — what constitutes a recovery?

The idea here is to assign degrees of UEs, where "[e]ach degree corresponds to a set of predicates which must be satsfied if recovery is to be considered successful".

Example 4.5.1 (Degrees of UE recovery)**.** Consider a browser encountering a problem while downloading a web page. The following degrees of recovery might be performed:

1. Display the webpage (possibly without some missing documents). If the HTML document was not successfully downloaded,
2. perform some guesswork to be able to display the HTML-code and other objects that were downloaded. If the HTML document was not downloaded at all,
3. display an error message or an empty document. In case of an internal error,
4. report the problem to the user and terminate the program.

The recovery attempts in the above example have different costs and results. Different situations might require a different order of the elements, or skip some or all of the recovery attempts. Two solutions are suggested. Alternative one is distributing multiple versions of the module, with different error handling strategies. Alternative two allows external access to recovery functions, so that the individual error handler may choose its own strategies.

4.5.2 Software Compartments

Software compartments were suggested for ADA in 1990[40] and revived for Java in 2000[60]. Litke, quoted by Robillard, states that compartments "(...) makes reasoning about program behaviour easier by reducing the complexity of relationships and makes modification of error-tolerating code easier." [40, p. 405]

The main point with software compartments is that a compartment, containing one or more modules, objects, packages etc, has a clearly defined exception interface. Components usually coincide with the components of the software architecture.

Compartments throw *abstract exceptions*. These exceptions are used by the compartment clients, and require knowledge of the compartment. Thus, abstract exceptions should not be propagated to a layer that does not have knowledge of the raising compartment.

Robillard and Murphy state the following guidelines for establishing exception interfaces:

1. Only use exceptions: Do not use return codes and global error codes for error handling.
2. Document exhaustive interfaces: Every exception that may be propagated, must be specified, both checked and unchecked.
3. Specify precise error semantics: Specify the precise semantics of every exception type. This is difficult and may not be neccessary in the case of hierarchial types, see the next item.
   (a) Design exception interfaces for change: Choose exception semantics that allows extension by subtypes. "This approach gives the client the option of either handling only the more general supertype exception or handling the more specific subtype exception."[60, p. 6]
4. Determine re-mappings for exceptions: The semantics of an exception must be updated during propagation to fit the surrounding system.
5. Avoid using system-defined exceptions as abstract exceptions.
6. Do not propagate abstract exceptions: Map to new a new exception when crossing compartment boundaries.
7. Do not raise abstract exceptions except in a compartment's entry points. Re-map intra-compartment exceptions at compartment boundaries.

Catch-all clauses[7] are used to ensure that only abstract exceptions is propagated across compartment boundaries. Exceptions caught here would indicate a programming error, and would be mapped to an exception like ProgrammingException[8]

How are abstract exceptions chosen to make room for later modifications? As established during my discussion of challenges with exception type subsumption[9], the new subtypes would risk being handled incorrectly.

[7] One of few acceptable uses, cf. Section 3.4.1
[8] See Section 2.3.4.
[9] See Section 3.2.5

Fortunately, the approach with software compartments reduce the number of exceptions[60] and promotes exceptions with ”wide” semantics.

Robillard and Murphy [60] does not answer how the system may correct itself when an exception occurs, only that a composition that raises an exception should be in a known state. Although the terminology is different, the paper that is presented next tries to solve this by wrapping every compartment with a safety facade, and placing exception handling in this wrapper.

Unfortunately, it is not clear if the author of the following paper knew of the idea of software compartments.

4.5.3 Safety Facades

Siedersleben [66] suggests grouping one or more components or modules into risk communities. Each community is wrapped by a safety facade (SF) that performs error handling. The safety facade is replacable. A composition manager may be tasked with selecting the appropriate SF.The risk communities are comparable to the compartments of the previous section.

Gathering the exception handling code in the safety facade simplifies replacing a component's exception handling strategies to fit various applications, and helps to maintain information hiding.

When deciding upon risk communities, a set of *abstract exceptions* is defined. These are the only exceptions that may be raised by a safety facade.

To its users, a safety facade implements a set of functions that may either return normally or fail safely. From Siedersleben [66]:

1. *Normal result:* The method was successful, which includes results that are errors from the view of the application, such as the withdrawal rejected for lack of funds. It is not disclosed to the caller whether an exception handling was required (...).

2. *Final and safe failure:* The method has finally failed; all repair attempts were unsuccessful and all measures for limiting the damage were performed. Further repair attempts are useless; the sole remaining option for the caller is to abort.

The downside with safety facades is that exceptions cannot easily be ignored. Exceptions are propagated to the safety facade, which makes returning control to the component difficult. For situations where ignoring an exception is acceptable, a `try..catch` block may need to be used in the component.

In principle, constructing a safety facade is simple. From [66],

1. Implement a new class *SF* that implements a suitable subset of the interfaces exported by the encapsulated component *C*.

2. When implementing SF call the original methods of *C* from a `try-catch` block.

The safety facade does not and should not have access to the protected components private data. Each component may instead provide an interface for diagnostics and repair (D&R interface). The interface may provide services like status information, resetting or shutting down the component. The actual services is tailored for each component, and some components may not need a D&R interface at all.

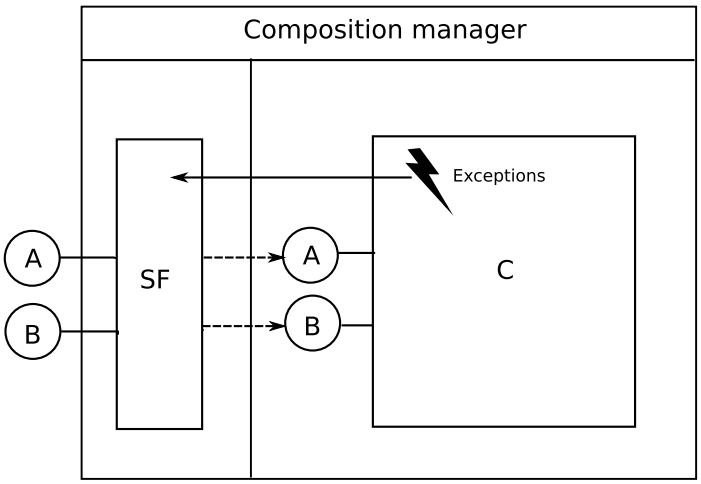

Figure 4.1: Safety Facade and Composition Manager[66].

In case of complex exceptions, D&R experts may handle groups of exceptions, e.g. an SQL-expert as well as a Network expert. The D&R expert objects is provided access to the D&R interface of the component they are designed for. Siedersleben [66] does not go into detail how these experts could be structured, whether they are provided by the component or simply is a way of structuring the safety facade.

Safety facades should be used in combination with the following rules: [66]

1. Distinguish Errors and Exceptions. To a file handling routine, EOF is an error, but is still a normal result. Exceptions are abnormal, and their handling is deferred to the next security facade.
2. Handle Errors Locally. Errors Are Never Propagated.
3. Return Errors Via Return Values (e.g. null). If This Is Not Possible, Return Errors as Checked Exceptions.
4. Signal Exceptions as Early as Possible.
5. Security Facades Catch Exceptions and Handle Them — Nobody Else.
6. Compose Components to Risk Communities.

Chapter 5

Safety Facades in Depth

At first glance, safety facades (SF) appear to solve a number of the exception handling problems I described earlier:

- Fewer global implications when throwing exceptions.
- The number of exceptions are reduced.
- No uncaught exceptions.
- Lower coupling between normal and exceptional code.
- Better information hiding.

These are so far only assumptions. Unfortunately, the paper by Siedersleben [66] is brief, presenting the highlights while leaving the reader to fill in the details. This chapter will describe these details, through

- examining the author's claims,
- comparing safety facades with existing methods,
- discussing the theoretical effect on identified exception handling challenges, and
- example code and design.

Although there is little information on safety facades, similar ideas have been floating around a long time, including software compartments[40, 60], mentioned earlier. Also, a version of safety facades has been implemented in the third-party J2EE library EL4J[2].

5.1 Introductory findings

Siedersleben [66] states three advantages and one disadvantage of using safety facades. These are a good place to start examining the effect of introducing SFs. Following a short elaboration on these claims, I will look into other properties of safety facades.

5.1.1 Siedersleben's claims

- + Information hiding is preserved.
- + Promotes reuse.
- + Simplifies the development process.
- - Raised exceptions terminate the current method.

Information hiding is preserved. The SF guarantees that exceptions are contained within the risk community, effectively hiding implementation details. Handling of the exceptions is also contained within the SF. Actions on a protected component is performed using the Diagnosis and Repair interface, further improving information hiding. Even if some implementation details is exposed by the component, its users is not affected.

Reuse of protected components is easier, because a components exception handling is easily replacable. Rather than providing specialized components, it may be enough to implement a specialized safety facade.

The development process is simplified. Adding complete exception handling may be postponed to a later stage during development. Usually, this is considered to be dangerous and is prone to leave null handlers (Section 3.4.1). With safety facades, the exception handling code is separated from normal code, reducing the number of catch clauses to be updated and keeping them close together. Siedersleben [66] states "It is possible to start with very simple exception handling, adding more handling when neccessary."

Resuming execution after an exception has been encountered is not trivial. The exception is raised to the safety facade, aborting the running method. At this point, the safety facade cannot return control to the point where the exception was raised. Using a D&R interface, the partially executed operation may be rolled back if applicable and retry.

To continue from where the exception was raised, Siedersleben [66] suggests wrapping a catch clause around the critical call. This catch clause may call an exception handling method in the safety facade. That requires a mechanism for a component C to retrieve its safety facade. The composition manager or the safety facade may provide C with a reference to the safety facade.

5.1.2 Design effects

I expect a number of changes in software structure:

- Longer distance from where an exception is raised to where it is handled.
- Handlers have reduced access to the data structure where the exception originated.
- Exception handling code becomes a separate component.
- Fewer exceptions propagated from each risk community.
- The process of making exceptional code is separated from normal code.
- "Cleanup code" is separated from "what to do next"-code.

5.1.3 Exceptions vs. return values

Robillard and Murphy [60] is determined that only exceptions should be used for error handling, leaving return codes and global error values to the history. The safety facade guidelines however allow return codes. Both sides may be considered. On one side, return codes

- are excellent for simple error reporting,
- requires less code,
- uses less resources than exceptions, and
- are familiar and well known to programmers.

On the other side, return values are prone to being ignored. In case of errors that must not be ignored, or for relaying complex information, checked exceptions are more appropriate. My opinion is that return codes are useful for returning error information, simply because they are easier to use, and they would probably be used even if disallowed by project guidelines.

In languages without checked exceptions, errors should probably be reported using return values, leaving exceptions for "truly exceptional situations or errors that should not be ignored."[44, p. 199]

5.1.4 Refactoring

What happens when safety facades are introduced by refactoring? Risk communities have much in common with software compartments (See Section 4.5.2). Refactoring programs with software compartments showed three resource intensive tasks[60],

1. Define exception interfaces, the "abstract exceptions" that are allowed to propagate between compartments. This is the most resource intensive task.
2. "Setting up exception guards at compartment boundaries to prevent unanticipated exceptions from escaping compartments", and
3. "tracking down meaningful exception sources to map them into the exception interfaces"

The first activity is a part of creating safety facades, even though methods may only complete successfully or fail safely. When a method fails, some information should be provided to its caller. How to do this, is not described by the original paper by Siedersleben.

Setting up exception guards is analogous when designing safety facades.

Tracking down exception sources is an important part of designing safety facades. Safety facades are responsible for any and all exceptions raised by its risk community. The act of mapping exceptions is downplayed. A safety facade favours handling exceptions rather than propagating them.

Robillard and Murphy [60] found refactoring to be fruitful on both simple and more complex programs. The biggest difference between safety facades

and compartments is the distribution and responsibilities of exception handling code. Software compartments ensure that architectural components exchange a small set of well defined exceptions. Safety facades additionally hides much of the exception handling from other components. This introduces additional costs with refactoring and moving existing exception handling code, as well as adding some transaction control and D&R interfaces.

The amount of refactoring depends on the size of the system and the resources available. In one extreme, the existing code could be handled as COTS or OSS, creating a safety facade as a wrapper with minimal change to the components internal exception handling.

Introducing safety facades

Is it possible to use safety facades as drop-in replacements of their protected components? Consider the situation where a safety facade is added to an existing system, wrapping a single component MyComponent. While safety facades export the same normal interfaces as its component, the exception interface is different. Initially, this does not appear to be a problem. In Example 5.1.1, both MyComponent and MyComponentSF implement the interface correctly. Note that type compatibility requires that the safety facade is a subclass of its protected component, or that they implement the same interface.

Example 5.1.1.
```
public interface IMyComponent {
        public void foo() throws IOException() { ... }
        public void bar() throws FileNotFoundException() { ... }
}

public class MyComponent implements IMyComponent {
        public void foo() throws IOException() { ... }
        public void bar() throws FileNotFoundException() { ... }

}                                                                    10

public class MyComponentSF implements IMyComponent {
        public void foo() { ... }
        public void bar() { ... }
}
```

Having considered this, safety facades do have different semantics and responsibilities than the protected component. Example 5.1.2 shows the interfaces of a simple component and its safety facade. How may MyComponent be replaced by its safety facade in ComponentUser.doWork()? At first glance, simply changing doWork() to make use of MyComponentSF is sufficient. The catch clause would no longer be triggered, while the normal code is unaffected.

Example 5.1.2.
```
public class MyComponent {
        public void foo() throws IOException() { ... }
        public void bar() throws FileNotFoundException() { ... }
}

public class MyComponentSF {
        public void foo() { ... }
```

```
        public void bar() { ... }
}
                                                                    10
public class ComponentUser {
        public void doWork() {
                MyComponent mc = new MyComponent();

                try {
                        mc.foo();
                } catch (IOException e) {
                        // Do something smart.
                }
        }                                                           20
}
```

There are however many situations where the safety facade cannot be a drop-in replacement of an existing component:

- The risk community consists of more than one component.
- Errors need to be reported. Unlike exceptions, errors is passed as return values or Checked Exceptions.
- Changes in the underlying component may result in different behaviour.

5.1.5 COTS and OSS

Commercial Off The Shelf software is usually distributed without source code. If using Open Source Software, the source code is available, but major changes to the software may still be out of the question. Local changes to OSS may be time consuming, and makes upgrading to newer releases harder.

In this situation, safety facades would have little control over a components internal exception handling. SFs are still very useful, because they provide protection against unexpected exceptions from the COTS software, as well as hides specific exception handling information from the surrounding system.

5.2 Selected challenges

Chapter 3 started with readability of exception handling code. Safety facades does not change the fundamental exception handling constructs, but does help exceptional code stand out by placing it in a separate component. During inspection, this component would have to be inspected as a separate component, rather than as a part of normal code. Safety facades appear to have a positive effect on many of the previously discussed challenges.

Code complexity may be reduced. A component's exception handling code is consolidated in the safety facade, rather than being placed in bits and pieces all ofter the component. On the other hand, it increases the distance from the exception occurrance to its handler.

Simply introducing better structured design of exception handling would be expected to have a positive impact on many of the challenges that have been discussed earlier.

5.2.1 Object Oriented Design

Previously I established that exception handling conflicts with a number of concepts in object-oriented design (See Section 3.2). Some of these are tackled directly by safety facades:

Partial States

A safety facade service will either succeed or fail, leaving no room for partial states. That having said, protected components may be in a partial state between the raising of an exception and it being caught and handled by the facade.

To achieve *complete exception specifications*, four pieces of information is needed:

Exception masking: The safety facade will either complete an operation, or report a failure. Exceptions of consequence to the underlying system will not be masked.

Exception consistency: The process of designing a system with safety facades makes explicit attention to abstract exceptions. This ensures consistent use of exception types.

Exception context: Not directly affected.

Object state: Safety facades aim to maintain a known state. If its protected components enter an unrecoverable state, it may shut down and refuse to provide further services.

Safety facades does not provide complete exception specifications. The problem is caused by the relationship between exception handling and the object-oriented paradigm. Design choices cannot solve this problem alone. What safety facades does, is to reduce the problem, rather than provide a complete and foolproof solution.

Abstraction

One issue with abstractions is that internal structures of a composition is unavailable for an external exception handler. An exception may need to be propagated between objects in a composition, to update its internal structure. Normally, information about the implementation should be hidden. In case of safety facades, it is possible to leak more information than would usually be recommended out of concern for increased cohesion.

Encapsulation

Encapsulation concerns how information and pointers to internal data may be abused to break information hiding. By using safety facades, exceptions that may expose information is caught and abstracted. In many cases, the safety facade attempts to handle errors rather than propagate them.

As earlier, this does not remove the problem, but appears to reduce it.

Modularity

As stated before, modularity is accomplished when a change in one software component does not require changes in another component. Safety facades affects modularity by grouping components into risk communities, providing a single access point to its protected components.

Inheritance

Inheritance problems does not appear to be directly affected by safety facades.

5.2.2 Implementation issues

Of the implementation issues, most were caused by lack of education, human errors and shortcuts. I would propose that many of these problems would disappear with education and some structure in designing of the exception handling.

Some of the problems *are* affected by safety facades. First, one would expect the number of null handlers to be reduced. Empty catch clauses and catch-all clauses are usually used to postpone or ignore exceptions. Where safety facades are used, it is less of a problem to introduce null handlers — just place them in a safety facade. These are easy to find and correct later.

Inconsistent exception semantics, disclosure of information or uncaught exceptions are effectively contained within a safety facade. These situations may still cause trouble, but the exception types are enclosed within the risk community where they are thrown.

The last problem presented in Section 3.4, was *Propagation from within handlers*. Exceptions may be raised from within `catch..finally` blocks. It is critical for a safety facade to avoid unintended exception propagation.

5.2.3 Development Process Integration

Børretzen et al. [18] introduced safety activities like HAZOP and FMEA into the RUP development process. Safety facades would benefit from being used in conjunction with these methods. An important early activity when designing the risk communities, is finding the abstract exceptions. Techniques like HAZOP and FMEA are excellent for finding exceptions.

Safety facades may be designed and implemented iteratively. At an early stage, risk communities are identified along with the software architecture. At later stages, exception handling functionality can be implemented in iterations.

5.3 Example use

To examine the properties of safety facades further, some experimental implementations will be presented. A version of safety facades have been a part of EL4J, a third-party J2EE library, since late 2005. This implementation will also be examined.

5.3.1 The StatusMonitor

A very simple safety facade is presented in the Status Monitor program. For source code, see Appendix A.1. In this example, a rudimentary safety facade, the WebMonitorSF (line 46), is presented. No D&R interface is used, and the exception handling is limited to an unintelligent "make five attempts".

Notice that the I/O class does not contain any exception handling. If an I/O error occurs, an IOException is thrown. This would likely signal a problem with connecting to the web server. If an exception is caught by the safety facade, the test will be tried again. This may reduce false positives caused by local network problems.

If the web page is downloaded successfully, the http return code is 200. Other possibilities includes the 500 Server side error, 404 Resource not found, and 403 Permission denied. In our case, these signal a server side error.

5.3.2 EL4J

EL4J is a third party open source J2EE[1] library made by ELCA[2].

EL4J provides a number of exception handlers[22, p. 71]:

- RethrowExceptionHandler: Forwars the exception to the caller.
- SimpleLogExceptionHandler: Logs the exception and its source.
- SequenceExceptionHandler: Invokes one exception handler after another until one succeeds.
- RetryExceptionHandler: Retries the same invocation several times.
- RoundRobinSwappableTargetExceptionHandler: Iterates over a list of different exception handlers. Run time configurable.

Using one of the existing exception handlers, it is trivial to define a safety facade without writing actual code.

In addition, EL4J has a *context exception handler*, that decides exception handling based on the context.

As a freely available J2EE-extension, the EL4J may be used in any J2EE-project.

[1] http://java.sun.com/javaee/
[2] http://www.elca.ch/

Chapter 6

Conclusion

In this thesis, I presented the background, and the difficulties associated with exception handling. Then I looked at possible problem solutions, before examining safety facades.

The first part of the thesis presented a large body of research on challenges with exceptions. The challenges spanned from simple implementation pitfalls, to more serious conflicts between exception handling and the object-oriented paradigm.

A body of proposed solutions were also presented. Out of these, safety facades was chosen for further study.

Safety facades are simple in principle. Services provided through safety facades will either complete normally, or fail safely. Exception handling is relocated from the components to their safety facades. Different contexts have use for different exception handling. This is accomplished by replacing the safety facade.

I have also shown how safety facades have a positive effect on many of the problems that is introduced by exception handling code. Most notably, safety facades is a simple concept to understand and is one of few solutions that add structured thinking to exception handling.

Unlike a number of other solutions, security facades does not require updates to the programming language or special tools. This is an important advantage..

6.1 Further work

A logical next step is to perform empirical testing on groups and individuals. There are a number of interesting questions, including

- Is the method easy to understand and use?
- How is the software structure affected?
- Are there measurable effects on code complexity?
- Is safety facade code easier to read?

The experiments could contain refactoring, making modifications to code with and without safety facades, and designing a system from scratch.

Safety facades should be a part of the development process. One of the early activities when using safety facades, is deciding risk communities and the abstract exceptions that should flow between them. This requires that exceptions are found at an early stage, and that safety facades is included early in the design phase. Some work has been done to add safety activities to RUP. Similarly, safety facades may be added to RUP or a similar development process as well.

The EL4J framework appears very interesting. Unfortunately, I did not discover it until the last week before the deadline for this thesis. Exception handling in EL4J is part of the architecture.

Appendix A

Code examples

A.1 StatusMonitor.java

The StatusMonitor is a simple, though a bit contrived application created to demonstrate how a safety facade might be used.

```
import java.net.HttpURLConnection;
import java.net.URL;
import java.io.IOException;
import java.util.Date;
import java.text.SimpleDateFormat;

/**
 * StatusMonitor is a simple, contrived, application that monitors
 * the status of a web site. A safety facade is used to
 * manage errors reported by a class responsible for performing
 * network I/O.
 */
public class StatusMonitor {
        public final static int ONE_MINUTE = 60000;
        public final static String WEB_PAGE_URL = "http://www.example.com/";

        WebMonitorSF msf;

        public StatusMonitor(String url) {
                msf = new WebMonitorSF(url);
        }

        public void start() {
                while(true) {
                        String state = msf.test() ? "up" : "DOWN";
                        System.out.println(getDateTime() + " " +
                                        WEB_PAGE_URL + " is " + state);
                        try {
                                Thread.sleep(ONE_MINUTE);
                        } catch (InterruptedException e) { }
                }
        }

        public String getDateTime() {
                return "[ " + new SimpleDateFormat("yyyy.MM.dd HH:mm:ss")
                        .format(new Date()) + " ] ";
        }

        public static void main(String[] args) {
                StatusMonitor m = new StatusMonitor(WEB_PAGE_URL);
                m.start();
        }

}

class WebMonitorSF {
        WebMonitor m;

        public WebMonitorSF(String url) {
                m = new WebMonitor(url);
        }

        public boolean test() {
                int attempts = 0;
                while(attempts < 5) {
                        try {
```

```
                        return m.test();
                    } catch (IOException ioe) {
                        // The exception is ignored,
                        // we only care about
                        // working vs not working.
                    }
                    attempts++;
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) { }
                }
                return false;
        }
}

class WebMonitor {
        String urlstr;

        public WebMonitor(String url) {
                // Todo: Validate url.
                this.urlstr = url;
        }

        /**
         * Perform a test by downloading the specified URL.
         *
         * return true if response code 200, else false.
*/
        public boolean test() throws IOException {
                URL url = new URL(urlstr);
                HttpURLConnection http =
                        (HttpURLConnection) url.openConnection();
                http.setConnectTimeout(2000);

                http.connect();
                http.getContent();

                if(http.getResponseCode() == 200) {
                        return true;
                } else {
                        return false;
                }
        }
}
```

Bibliography

[1] Exception safe programming. http://digitalmars.com/d/exception-safe.html (accessed 2007-04-27). Documentation for the D programming language.

[2] El4j. URL `http://el4j.sourceforge.net/`.

[3] Alfred A. Aho, Ravi Sethi, and Jeffrey D. Ullmann. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Boston, USA, 2003. ISBN 0-201-10194-7.

[4] Matthew Allen and Susan Horwitz. Slicing java programs that throw and catch exceptions. In *PEPM '03: Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 44–54, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-667-6.

[5] Peter Amey and Roderick Chapman. Industrial strength exception freedom. In *SIGAda '02: Proceedings of the 2002 annual ACM SIGAda international conference on Ada*, pages 1–9, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-611-0. doi: http://doi.acm.org/10.1145/589451.589452.

[6] Tanveer Hussain Awan. Sources of variations in software inspections: An empirical research project. Master's project, Norwegian University of Science and Technology, Institute of Computer and Information Science, 2004.

[7] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, Boston, USA, 2003. ISBN 0-321-15495-9.

[8] Andrew P. Black. *Exception Handling: The Case Against*. D.Phil thesis, University of Oxford, January 1982.

[9] Joshua Bloch. *Effective Java*. The Java Series. Sun Microsystems, Inc., Mountain View, CA, USA, 2001. ISBN 0-201-31005-8.

[10] IEEE Standards Board. *IEEE Standard Glossary of Software Engineering Technology*. The Institute of Electrical and Electronics Engineers, 1990. ISBN 1-55937-067-X.

[11] Marco Brambilla, Stefano Ceri, Sara Comai, and Christina Tziviskou. Exception handling in workflow-driven web applications. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 170–

179, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-046-9. doi: http://doi.acm.org/10.1145/1060745.1060773.

[12] Eric J. Braude. *Software engineering : an object-oriented perspective*. Wiley, 2001. ISBN 0-471-32208-3.

[13] Pete Broadwell and Emil Ong. A comparison of static analysis and fault injection techniques for developing robust system services. Technical report, University of California, Berkeley, may 2003. http://www.cs.berkeley.edu/ pbwell/papers/saswifi.pdf.

[14] David Budgen. *Software Design*. Addison Wesley, 2003. ISBN 978-0201722192.

[15] Peter A. Buhr and Roy Krischer. *Bound Exceptions in Object-Oriented Programming*, volume 4119 of *Lecture Notes in Computer Science*, pages 1–21. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-37443-5.

[16] Peter A. Buhr and W. Y. Russel Mok. Advanced exception handling mechanisms. *IEEE Transactions on Software Engineering*, 26(9):820–836, September 2000.

[17] Gary N. Bundy and Diane E. Mularz. Error-prone exception handling in large ada systems. In *Ada-Europe '93: Proceedings of the 12th Ada-Europe International Conference*, pages 153–170, London, UK, 1993. Springer-Verlag. ISBN 3-540-56802-6.

[18] Jon Arvid Børretzen, Tor Stålhane, Torgrim Lauritsen, and Per Trygve Myhrer. Safety activities during early software project phases. In *Proceedings of the Norwegian Informatics Conference (NIK'04)*. Tapir forlag, November 2004. ISBN 82-519-2004-3.

[19] Robert N. Charette. Why software fails. *IEEE Spectrum*, 42(9):42–49, Sept 2005.

[20] Raymond Chen. http://blogs.msdn.com/oldnewthing/archive/2005/01/14/352949.aspx (accessed 2007-04-27), January 2005.

[21] *Standard ECMA-334: C# Language Specification*. ECMA International, Geneva, third edition edition, December 2002. URL `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf`.

[22] ELCA, Switzerland, 1.3.6 edition, February 2007. URL `http://el4j.sourceforge.net/docs/pdf/ReferenceDoc.pdf`.

[23] Fernando Castor Filho, Patrick H. S. Brito, and Cecília Mary F. Rubira. A framework for analyzing exception flow in software architectures. In *WADS '05: Proceedings of the 2005 workshop on Architecting dependable systems*, pages 1–7, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-124-4. doi: http://doi.acm.org/10.1145/1083217.1083221.

[24] Fernando Castor Filho, Patrick H. S. Brito, and Cecília Mary F. Rubira. Specification of exception flow in software architectures. *The Journal of Systems and Software*, 79(10):1397–1418, oct 2006.

[25] Fernando Castor Filho, Cecília Mary F. Rubira, Raquel de A. Maranhão Ferreiera, and Alessandro Garcia. *Aspectizing Exception Handling: A Quantitative Study*, volume 4119 of *Lecture Notes in Computer Science*, pages 255–273. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-37443-5.

[26] Chen Fu and Barbara G. Ryder. Navigating error recovery code in java applications. In *eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 40–44, New York, NY, USA, 2005. ACM Press. ISBN 1-58113-000-0. doi: http://doi.acm.org/10.1145/1117696.1117705.

[27] Alessandro F. Garcia and Cecília M. F. Rubira. A comparative study of exception handling mechanisms for building dependable object-oriented software. Technical report, Newcastle University, 2000.

[28] Alessandro F. Garcia and Cecília M. F. Rubira. An architectural-based reflective approach to incorporating exception handling into dependable software. In Alexander Romanovsky, editor, *Advances in exception handling techniques*, volume 2022 of *Lecture Notes in Computer Science*, pages 189–206. Springer Berlin / Heidelberg, 2001. ISBN 3-540-41952-7.

[29] Alessandro F. Garcia, Delano M. Beder, and Cecília M. F. Rubira. An exception handling software architecture for developing fault-tolerant software. In *Proceedings on the 5th IEEE International Symposium on High Assurance Systems Engineering*, pages 311–320, Los Alamitos, CA, USA, 2000. IEEE Computer Society. ISBN 0-7695-0927-4.

[30] John B. Goodenough. Exception handling: issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, 1975. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/361227.361230.

[31] Nick Guerra. Why does fxcop warn against catch(exception)? http://blogs.msdn.com/fxcop/archive/2006/06/14/631923.aspx (2006-10-08), July 2006.

[32] Oddleif Halvorsen and Oystein Haugen. Proposed notation for exception handling in uml 2 sequence diagrams. In *Australian Software Engineering Conference (ASWEC'06)*, pages 29–40, Los Alamitos, CA, USA, 2006. IEEE Computer Society. ISBN 0-7695-2551-2. doi: http://doi.ieeecomputersociety.org/10.1109/ASWEC.2006.41.

[33] Charles Howell and Gary Vecellio. Experiences with error handling in critical systems. In Alexander Romanovsky, editor, *Advances in exception handling techniques*, volume 2022 of *Lecture Notes in Computer Science*, pages 181–188. Springer Berlin / Heidelberg, 2001. ISBN 3-540-41952-7.

[34] Valérie Issarny and Jean-Pierre Banâtre. Architecture-based exception handling. In *Proceedings on the 34th Annual Hawaiian International Conference on System Sciences*, volume 9, pages 9058–9067, Los Alamitos, CA, USA, 2001. IEEE Computer Society. ISBN 0-7695-0981-9.

[35] Bijay K. Jayaswal and Peter C. Patton. *Design for Trustworthy Software*. Pearson Education, Inc., Upper Saddle River, NJ 07458, USA, 2006. ISBN 0-13-187250-8.

[36] David M. Kean. Code analysis (team system) msdn chat roundup. http://blogs.msdn.com/fxcop/archive/2007/02/20/code-analysis-team-system-msdn-chat-roundup.aspx, February 2007.

[37] Jun Lang and David B. Stewart. A study of the applicability of existing exception-handling techniques to component-based real-time software technology. *ACM Transactions on Programming Languages and Systems*, 20 (2):274–301, mar 1998.

[38] K. Rustan M. Leino and Wolfram Schulte. Exception safety for c#. In *Proceedings on the 2nd International Conference on Software Engineering and Formal Methods*, pages 218–227, Los Alamitos, CA, USA, September 2004. IEEE Computer Society. ISBN 0-7695-2222-X. URL `http://research.microsoft.com/specsharp/papers/krml135.pdf`.

[39] Martin Lippert and Cristina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 418–427, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-206-9. doi: http://doi.acm.org/10.1145/337180.337229.

[40] John D. Litke. A systematic approach for implementing fault tolerant software designs in ada. In *TRI-Ada '90: Proceedings of the conference on TRI-ADA '90*, pages 403–408, New York, NY, USA, 1990. ACM Press. ISBN 0-89791-409-0. doi: http://doi.acm.org/10.1145/255471.255565.

[41] Donna Malayeri and Jonathan Aldrich. Practical exception specifications. In Christophe Dony, Jørgen Lindskov Knudsen, Alexander Romanovsky, and Anand Tripathi, editors, *Advanced Topics in Exception Handling*, volume 4119 of *Lecture Notes in Computer Science*, pages 201–220. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-37443-5.

[42] Cheng-Ying Mao and Yan-Sheng Lu. Improving the robustness and reliability of object-oriented programs through exception analysis and testing. In *Proc. 10th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS'05)*, pages 432–439, 2005.

[43] Roy A. Maxion and Robert T. Olszewski. Eliminating exception handling errors with dependability cases: A comparative, empirical study. *IEEE Transactions on Software Engineering*, 26(9):888–906, sep 2000.

[44] Steve McConnell. *Code Complete*. Microsoft Press, 2004. ISBN 978-0735619678.

[45] Tim McCune. Exception-handling antipatterns. http://today.java.net/pub/a/today/2006/04/06/exception-handling-antipatterns.html (5. Nov. 2006), 2006.

[46] Bertrand Meyer. *Object-oriented software construction*. Prentice Hall, 1997. ISBN 0-13-629155-4.

[47] Anna Mikhailova and Alexander Romanovsky. Supporting evolution of interface exceptions. In *Advances in Exception Handling Techniques*, number 2022 in Lecture Notes in Computer Science, pages 94–110, 2001. ISBN 3-540-41952-7.

[48] Robert Miller and Anand Tripathi. The guardian model and primitives for exception handling in distributed systems. *IEEE Software*, 30(12):1008–1022, Nov–Dec 2004. ISSN 0098-5589.

[49] Robert Miller and Anand Tripathi. Issues with exception handling in object-oriented systems. In *ECOOP'97 — Object Oriented Programming*, number 1241 in Lecture Notes in Computer Science, pages 85–103. Springer Berlin / Heidelberg, March 1997. doi: 10.1007/BFb0053375. URL `http://www.springerlink.com/content/t566514357105m6t`.

[50] Arup Mukherjee and Daniel P. Siewiorek. Measuring software dependability by robustness benchmarking. *IEEE Transactions on Software Engineering*, 23(6):366–378, June 1997.

[51] Dr. Andreas Müller and Geoffrey Simmons. Exception handling: Common problems and best practice with java 1.4. In *Net.ObjectDays 2002*, 2002. URL `http://www.old.netobjectdays.org/pdf/02/papers/industry/1430.pdf`.

[52] Hoa Dang Nguyen and Magnar Sveen. Rules for developing robust programs with java exceptions. Technical report, Norwegian University of Science and Technology, 2003.

[53] Heejung Ohe and Byeong-Mo Chang. An exception monitoring system for java. In N. Guelfi, editor, *Rapid Integration of Software Engineering Techniques*, volume 3475 of *Lecture Notes in Computer Science*, pages 71–81. Springer Berlin / Heidelberg, 2004. ISBN 978-3-540-25812-4.

[54] Jiantao Pan. The dimensionality of failures – a fault model for characterizing software robustness. Technical report, Carnegie Mellon University, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA 152213, USA. URL `http://www.ece.cmu.edu/~koopman/ballista/ftcs99_pan/index.html`.

[55] Derek A. Park and Stephen V. Rice. A framework for unified resource management in java. In *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 113–122, New York, NY, USA, 2006. ACM Press. ISBN 3-939352-05-5. doi: http://doi.acm.org/10.1145/1168054.1168070.

[56] D. L. Parnas and H. Würges. Response to undesired events in software systems. pages 437–446, 1976.

[57] Gergely Pintér and István Majzik. Modeling and analysis of exception handling by using uml statecharts. In *Scientific Engineering of Distributed Java Applications*, number 3409 in Lecture Notes in Computer Science, pages 58–67. Springer Berlin / Heidelberg, February 2005. ISBN 978-3-540-25053-1. URL `http://www.springerlink.com/content/610ru59wgh15hh15`.

[58] James Reason. *Human Error*. Cambridge University Press, Cambridge, England, 1990. ISBN 0521314194.

[59] Darrel Reimer and Harini Srinivasan. Analyzing exception usage in large java applications. In *ECOOP 2003 Workshop proceedings*, pages 10–19, July 2003. URL `http://homepages.cs.ncl.ac.uk/alexander.romanovsky/home.formal/EHOOS-report.pdf`.

[60] Martin P. Robillard and Gail C. Murphy. Designing robust java programs with exceptions. *SIGSOFT Softw. Eng. Notes*, 25(6):2–10, 2000. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/357474.355046.

[61] Martin P. Robillard and Gail C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. on Software Engineering and Methodology*, 12(2):191–221, 2003.

[62] Ioana Rus, Sija Komi-Sirvio, and Patricia Costa. Software dependability properties – a survey of definitions, measures and techniques. Technical report, NASA High Dependability Computing Program, jan 2003. http://www.hdcp.org/.

[63] Barbara G. Ryder and Mary Lou Soffa. Influences on the design of exception handling: Acm sigsoft project on the impact of software engineering research on programming language design. *SIGPLAN Not.*, 38(6):16–22, 2003. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/885638.885644.

[64] Jim Shore. Fail fast. *IEEE Software*, 21(5):21–25, Sept–Oct 2004. ISSN 0740-7459.

[65] Johannes Siedersleben. Errors and exceptions — rights and responsibilities. Talk at ECOOP'2003, July 2003. URL `http://homepages.cs.ncl.ac.uk/alexander.romanovsky/home.formal/Johannes-talk.pdf`.

[66] Johannes Siedersleben. *Errors and Exceptions — Rights and Obligations*, volume 4119 of *Lecture Notes in Computer Science*, pages 275–287. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-37443-5.

[67] Saurabh Sinha and Mary Jean Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering*, 26(9):849–871, sep 2000.

[68] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 2000. ISBN 0-201-700735.

[69] Walter F. Tichy. Should computer scientists experiment more? *IEEE Computer*, 31(5):32–40, 1998. ISSN 0018-9162.

[70] Marko van Dooren and Eric Steegmans. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 455–471, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094847.

[71] Bill Venners. Failure and exceptions. *Artima.com*, sep 2003. URL `http://www.artima.com/intv/solid.html`. Interview with James Gosling.

[72] Bill Venners. The trouble with checked exceptions. *Artima.com*, aug 2003. URL `http://www.artima.com/intv/handcuffs.html`. Interview with Anders Hejlsberg.

[73] Westley Weimer. Exception-handling bugs in java and a language extension to avoid them. In Christophe Dony, Jørgen Lindskov Knudsen, Alexander Romanovsky, and Anand Tripathi, editors, *Advanced Topics in Exception Handling*, volume 4119 of *Lecture Notes in Computer Science*, pages 22–41. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-37443-5.

[74] Westley Weimer and George C. Necula. Finding and preventing runtime error handling mistakes. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 419–431, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-831-9. doi: http://doi.acm.org/10.1145/1028976.1029011.