



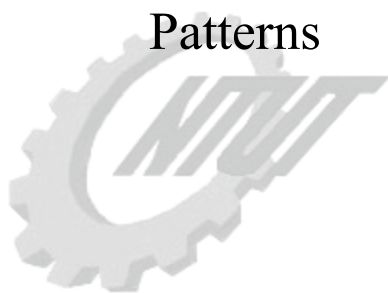
國立臺北科技大學

機電科技研究所博士班

博士學位論文

爪哇例外處理：模型、重構、與樣式

Java Exception Handling: Models, Refactorings, and
Patterns



研究生：陳建村

指導教授：鄭有進、謝金雲

中華民國九十七年九月

摘要

論文名稱：爪哇例外處理：模型、重構、與樣式

頁數：139 頁

校所別：國立台北科技大學機電科技研究所博士班（資訊組）

畢業時間：九十七學年度第一學期

學位：博士

研究生：陳建村

指導教授：鄭有進、謝金雲

關鍵字：例外處理模型、重構、樣式、爪哇語言、軟體設計

妥善的例外處理可以增強軟體的強健度。雖然例外處理已經被研究了數十年，然而對於軟體開發人員而言，例外處理依舊是一件困難且不易完成的工作。本研究的目的是在於探討實務上例外處理設計的困難點以及在爪哇語言中使用「檢查例外」與「非檢查例外」所造成的困擾，並提出解決方案。

遵循「把大問題切割成若干較小的問題，再個個擊破」的基本原理，我們認為完善的例外處理設計可藉由分階段完成較小的例外處理設計活動來達成。本論文的研究方法包含三個部分：我們首先定義四個例外處理強健度等級，以作為例外處理設計的目標；我們並分析達成不同強健度等級所需的例外處理基本操作及其成本。接著，我們整理例外處理程式的「壞味道」，並應用軟體重構技術來移除這些壞味道以達到改善軟體系統強健度的目標。最後，我們分析爪哇語言例外處理機制在設計上可能造成例外資訊遺失的問題，並提出一個以樣式為基礎的設計，在不改變爪哇語言的前提下，解決此一問題。

本研究所提出的方法已在實際的軟體專案中被實踐並驗證其可行性。本論文探討其中兩個案例，並簡介兩個依據本研究所開發的例外處理輔助工具。

ABSTRACT

Title : Java Exception Handling: Models, Refactorings, and Patterns

Pages : 139

School : National Taipei University of Technology

Department : Graduate Institute of Mechanical and Electrical Engineering

Time : September, 2008

Degree : Ph.D.

Researcher : Chien-Tsun Chen

Advisor : Yu Chin Cheng, Chin-Yun Hsieh

Keywords : Exception handling models, refactoring, patterns, Java, software design

Exception handling can improve robustness, which is an important quality attribute of software. Although exception handling has been studied for decades, it remains a difficult task to get right. The purpose of this research is to investigate the difficulty of exception handling design in practice and the problem of the use of checked and unchecked exceptions in the Java language. Based on our finding, solutions are proposed.

Following the general practice of solving a complex problem with divide and conquer, we believe that exception handling design problems are best tackled by conducting a series of small design activities. The research methods of this dissertation are threefold: first, we propose a robustness model with four levels of goals for exception handling. Primitive operations of exception handling and their implementation costs are presented. Second, we introduce exception handling smells that hinder the achievement of the robustness goals and propose exception handling refactorings to eliminate the smells. Finally, we analyze a problem of the Java exception

handling mechanism which may cause the lost of exception information; A pattern-based solution is proposed to solve the problem without altering the Java language.

The proposed approaches have been practiced in several real world software projects. To evaluate the usefulness and effectiveness of the research, two case studies and two exception handling supporting tools are presented.

誌謝

CONTENTS

摘要	ii
ABSTRACT	iii
誌謝	v
CONTENTS	vi
LIST OF TABLES	viii
LIST OF FIGURES	ix
1 Chapter 1 INTRODUCTION	1
1.1 Motivation	1
1.2 Research Methods	2
1.3 Organization of the Dissertation	4
2 Chapter 2 BACKGROUND	5
2.1 Exceptions and Exception Handling Mechanisms	6
2.2 Views on Exception Handling	9
2.3 Exception Handling in Java	12
3 Chapter 3 A STAGED ROBUSTNESS MODEL	16
3.1 Goals of Exception Handling	16
3.2 Checked and Unchecked Exceptions	20
3.2.1 Programming to an interface regarding exceptions	21
3.3 Exception Handling Primitives and Costs	23
3.4 Documenting Exception Handling Goals	28
4 Chapter 4 EXCEPTION HANDLING REFACTORING	32
4.1 Refactoring	32
4.2 Bad Smells	34
4.3 Refactorings	36
5 Chapter 5 A PATTERN-BASED SOLUTION TO THE JAVA EXCEPTION OVERRIDDEN PROBLEM	57
5.1 Problem Description	57
5.2 Exception Handling Patterns	63
5.3 Summary of the patterns	91
6 Chapter 6 APPLICATIONS	92
6.1 The Robustness Model in Practice	92
6.1.1 The first scenario	95
6.1.2 The second scenario	97

6.1.3	The third scenario	97
6.1.4	Cost analysis of the example	98
6.1.4.1	Summary	100
6.1.4.2	Characteristics of the first scenario.....	102
6.1.4.3	Characteristics of the second scenario	102
6.1.4.4	Characteristics of the third scenario.....	104
6.2	A Case Study of Exception Handling Refactoring	105
6.2.1	The credit scoring system	105
6.2.2	Strategies to apply the refactorings.....	106
6.2.3	EH refactoring in action.....	109
6.2.3.1	From G0 to G1	109
6.2.3.2	From G1 to G2	111
6.2.3.3	From G2 to G3	111
6.2.4	Results.....	111
6.2.5	Cost-effectiveness analysis	115
6.2.5.1	Code analysis	115
6.2.5.2	Time analysis	115
6.2.5.3	Cost analysis	116
6.3	An Exception Handling Refactoring Tool	116
6.3.1	The Eclipse Quick Fix for unhandled checked exceptions.....	117
6.3.2	Tool design.....	119
6.4	A Robustness Visualization Tool	121
7	Chapter 7 RELATED WORK	122
7.1	The Robustness Model of Exception Handling	122
7.2	Exception Handling Refactoring	122
7.3	Exception Handling Patterns	124
8	Chapter 8 CONCLUSION	126
8.1	Contribution.....	126
8.2	Future Work	127
	References.....	128
	VITA	133

LIST OF TABLES

Table 3.1: Robustness levels of a component and its effect on the program after encountering an exception.....	17
Table 3.2: Applicability for the level of robustness.....	18
Table 3.3: Exception handling primitive operations.....	24
Table 3.4: Exception handling cost analysis	26
Table 3.5: Robustness level of the <code>writeFile</code> method documented with Java annotations and with C# custom attributes.....	29
Table 5.1: The behavior of <code>signalIfError</code> for each state.....	86
Table 6.1: Cost analysis of the three scenarios.....	99
Table 6.2: The cost value assigned to each primitive operation.....	100
Table 6.3: Statistics of the top three most frequently occurred bugs attributed to exception handling in the year of 2005.....	108
Table 6.4: The number of failures reported before and after refactoring	113
Table 6.5: Code analysis of the refactoring case study	114
Table 6.6: Time spent in the refactoring case study	114
Table 6.7: Cost-benefit analysis in terms of money saved.....	115

LIST OF FIGURES

Figure 3.1: Upgrading and degrading exception handling goals	20
Figure 3.3: The source code of robustness documentation using (a) Java’s annotations, and (b) C#’s custom attributes	30
Figure 5.2: An execution path showing the lost of exception problem.	59
Figure 5.6: Patterns applied in <code>CleanerUtility</code> and <code>Cleaner</code> classes.....	68
Figure 5.9: <i>Safety Method</i> and <i>Safety Command</i> applying <i>Collecting Parameter</i>	71
Figure 5.14: The <code>Thrower</code> class.	81
Figure 5.15: The revision of exception handling with retry.....	83
Figure 5.16: A state diagram for the <code>Thrower</code> class.....	85
Figure 5.17: A class diagram of the <code>Thrower</code> and its states.	85
Figure 5.18: Relationships between active, subsequent, and raw exceptions.....	87
Figure 5.19: Source code of (a) <code>ICollection</code> and (b) <code>EHException</code>	89
Figure 5.21: A pattern language representation of the proposed patterns.....	91
Figure 6.2: Three scenarios of exception handling evolution.....	94
Figure 6.5: System context diagram of the credit scoring system (CSS)	106
Figure 6.6: The check method before refactoring.....	110
Figure 6.7: The check method after refactoring for G1	110
Figure 6.8: The refactored check method for G3.....	112
Figure 6.9: The statement throwing an unhandled exception is underlined	117
Figure 6.10: Exception handling proposals provided by Eclipse	118
Figure 6.11: Code after applying the “Surround with try/catch” proposal	118

Figure 6.12: Conceptual model for the exception handling refactoring tool	120
Figure 6.13: The code template when choosing “Add top-level catcher” policy.....	120
Figure 6.14: A sequence diagram with robustness information	121

Chapter 1 INTRODUCTION

1.1 Motivation

Exceptions are the main abstraction of errors and failures in many programming languages. Correspondingly, exception handling is a program's means to respond to the detected error, preventing it from surfacing as a system-level failure. Done well, an exception handling design can improve system robustness and increase system dependability.

Granted its obvious merits, most software developers will agree that exception handling design is a difficult issue. Many tried-and-true best practices are published in the literature, but not without disputes. One of the most notable instances is Java's separation of checked and unchecked exceptions. While the language designers thoughtfully introduce it as a means to write solid programs [1][2], others have doubted its value [3][4].

But that is not all. Since exception handling design is an issue of robustness, which is a non-functional requirement, it is considered together with other competing and often conflicting project factors such as time-to-market, iterative and incremental design, maintainability, and so on. Given limited project resource, tradeoffs are unavoidable. On the one hand, with the current high interest on shortening product development time, it is small wonder that exception handling design often ends up on the losing side of a tradeoff decision—at least in the initial release. There is just not enough time. On the other hand, exception handling design cannot be ignored for long, either. The reason is that users generally demand good software quality in the long term. Occasional failures may be tolerated by the users while novelty lasts, inadequate exception handling design can eventually turn them away to competitors.

Moreover, Java programmers are easily led to ignore valuable exception information due to the well-known *Java exception overridden problem* [5]. The execution of a method can encounter multiple exceptions, including checked exception thrown by the invocation of other methods in the `try` clause, cleanup exceptions in the `finally` clauses, error handling exceptions in the `catch` clauses, and so on. Each of these exceptions carries valuable information: a cleanup exception can indicate resource leak; an error handling exception can indicate that the faulting component may be in an incorrect state; and so on. Properly preserved, the collective information carried by the raised exceptions can be instrumental in facilitating exception handling during development and for bug fixing during maintenance. However, the current Java exception handling mechanism makes it difficult to collect complete and unambiguous exception information when multiple exceptions are encountered. As a result, some debugging information may be lost and programmed exception handling based on exception information may not be possible.

So, the question is: how do you plan for Java exception handling design for long-termed robustness? Even if you will not be actively attacking the exceptions in the initial release, what must you do to ensure that a good exception handling design can still be achieved in future releases, and to do so in a controlled manner? How can you improve the robustness of existing software? And is it possible to solve the Java exception overridden problem without changing the Java language?

1.2 Research Methods

To answer the above questions, this research takes the following approaches: first, we identify a staged robustness model that contains four exception handling goals for guiding developers in planning an overall exception handling design for Java programs. The model divides exception handling design into clearly defined intermediate

milestones that are easily exercised and verified. We also discuss how Java checked and unchecked exceptions can fit into the robustness model, propose the primitive exception handling operations to implement the robustness model, and suggest a way to document the robustness model in source code to facilitate exception handling tool support.

Next, we apply refactoring to improve existing exception handling design. With the help of the staged robustness model that defines clear exception handling goals, up-front exception handling design becomes easier. However, like most software design problems, even if developers are requested to design with exception handling beforehand, it is very difficult to get the right design at the first shot. Therefore, improving exception handling design after software is constructed is necessary. We introduce exception handling smells that hinder the achievement of the goals and propose exception handling refactorings to eliminate the smells.

We then look for future improvement of the Java exception handling mechanism. Specifically, we apply the pattern approach to tackling the Java exception overridden problem. To facilitate the collection of complete and unambiguous exception information, we propose exception handling patterns that suggest: (1) collecting all exceptions thrown by exception handling activities such as error handling, fault handling, cleanup, and retrying; (2) designing an exception hierarchy to represent failures of these exception handling activities; and (3) attaching the collected exceptions to the declared exception of a method. In implementing the patterns, we establish well-defined meanings for Java exception classes, investigate possible Java language enhancements for exception handling, and develop a pattern-based utility to facilitate the implementation of exception handling.

Finally, case studies are conducted and tools are developed to evaluate the usefulness and effectiveness of the model and refactoring of exception handling.

1.3 Organization of the Dissertation

This dissertation is organized as follows. Chapter 2 provides background information of exception handling. Chapter 3 presents a staged robustness model, primitive operations to implement the model, and a method to document the model in source code. In Chapter 4, bad smells and exception handling refactorings are presented. We describe a new pattern-based approach to the Java exception overridden problem in Chapter 5. Applications derived from the dissertation are shown in Chapter 6. Related work is presented in Chapter 7. Chapter 8 concludes this dissertation.

Chapter 2 BACKGROUND

With the ubiquitous presence of software in computers of all scales and embedded systems found in vehicles, medical equipment and commercial electronic devices, our daily lives are becoming increasingly dependent on the smooth operations of software. A *failure* of software indicates that the service provided by the software departs from its functional specification. Software failures can have a wide range of ramifications from small annoyances of interrupted entertainments to major system breakdowns with grave consequences. Causes of failures can be traced to *faults* of software. Two main categories of faults are distinguished: *design faults* and *component faults* [6]. A design fault is a defect, or a bug, that is implanted into the software during design and implementation. For instance, omission to ensure proper initialization of a component before accessing its services is a design fault. A component fault is caused by the abnormal circumstances encountered by the component during normal interactions with other components or the environment. An example is a properly initialized communication link that becomes intermittently disconnected during a remote transaction. Unless prevented, faults turn into an *error*, a state that might lead a component into failure. An error is detected either by program logic such as *assertions*, *checks*, and *acceptance tests* or by a monitoring mechanism in the runtime environment such as resource usage checks performed by operation systems or virtual machines. Instances of errors include invoking a null object, stack overflow, reading a non-existing file, running out of memory and disk spaces, network connection timeouts, and so on.

2.1 Exceptions and Exception Handling Mechanisms

Exception is an abstraction of errors and failures. As such, an exception carries information about the errors or failures it represents. *Exception handling* is the programmed activities conducted among multiple collaborating components in a program in response to an exception. These activities are conducted by code units called *exception handlers* or *handlers*. A handler can be attached to a statement, a block, a method, an object, a class, a module, or an exception. The plan about how to choose or devise an exception to represent errors and failures and how to respond to an exception is called *exception handling design*. An *exception handling mechanism* (EHM) is a programming language's fundamental supports for dealing with exceptions, including *representation*, *definition*, *signaling*, *propagation*, *resolution*, and *continuation* of exceptions.

An important objective in the design of EHM is to keep separated the code for delivering normal functionalities and the code for dealing with exceptional conditions. In a language like C that lacks adequate EHM support, programs explicitly check each function call on the return code, data fields, flags, or global variables for finding out about the execution status. Since the checks and handling of anomalies are mixed with the logic for normal functionalities, code becomes cluttered and the program becomes incomprehensible. EHMs in modern programming languages provide better supports for separation of concerns in this regard. With exceptions, it is no longer needed to check every statement for erroneous state. Once the execution of a statement gives rise to an exception, the runtime environment automatically alters the program execution from a normal path to an exceptional path. This allows the programmer to structure the logic for exception handling in appropriate exception handling constructs without cluttering the program.

Exception representation is the way that a programming language expresses an exception internally. It defines an *exception context* which contains information explicitly passed by the exception creator or implicitly passed by the language runtime. An exception can be represented as a *symbol*, a *data object*, or a *full object* [7]. Exceptions defined as symbols are strings or numbers. Data-object exceptions are primarily used to hold error and failure information only. Full-object exceptions directly encapsulate signaling, propagation, and continuation behavior of exceptions in the class definition.

Programmers use *exception definition* to define exceptions in a program. If exceptions are represented as symbols, new exceptions are defined as strings or numbers. If exceptions are data objects and full objects, a class is used to define an exception. Some languages allow any class to be exceptions while others require that only particular types of classes can be exceptions. For example, in C++ any class can be exceptions but in Java only classes which directly or indirectly inherit from the `Throwable` class can be exceptions.

An *exception occurrence* is an instance of an exception. The instruction to explicitly transmit an exception occurrence to the exception receiver is called *throwing*, *signaling*, *raising*, or *triggering* an exception. The sender of the exception is called *signaler*; the receiver of an exception is called the *exception target*, or *target*. An explicit throw instruction creates a *synchronous exception*, which is a direct result of performing the instruction. Conversely, an *asynchronous exception* can occur at any time in spite of the program statements under execution. Asynchronous exceptions can be produced by the runtime environment upon encountering an internal error or by stopping or suspending a thread.

If an exception is signaled and not coped with locally, the exception can be propagated to the caller of the signaling method. *Exception propagation* can be *explicit* or *implicit* (or *automatic*). In the former, a receiver must explicitly re-throw an unhandled received exception for further propagation; in the latter, an unhandled exception is automatically propagated. An exception is *internal* if it is not propagated; otherwise it is *external*. Some languages require propagated exceptions to be *declared* in signatures while others allow exceptions to be propagated without declaration.

Exception resolution or *handler binding* is a process of finding a suitable handler in the target, which is resolved by static scoping at compiler-time, dynamic invocation chain at runtime, or both. There are two methods to dynamically find a handler: *stack unwinding* and *stack cutting*. Stack unwinding pops the stack frames to search for the matching exception handler while stack cutting maintains a list of registered exception handlers and looks up the list for a suitable exception handler. While stack unwinding incurs no overhead in the normal execution, a significant overhead is involved in the exceptional execution. In contrast, stack cutting requires some housekeeping overhead to register and deregister exception handlers in the normal execution but incurs only a relatively minor overhead in the exceptional execution.

An *exception continuation* or *exception model* specifies the execution flow after an exception handler returns its control. In the *termination model* or the *nonresumptive model*, the execution of the component is terminated. A variation of the termination model is the *retry model* where the original execution is terminated and then the component is engaged for execution again. In the *resumption model*, the execution continues from the location or the next location where the exception was raised. Although program languages can support more than one model, mainstream programming languages such as C++, Java, and C# favor the termination model for its

simplicity.

An exception becomes *active* once it is raised. At any given stack frame, at most one exception is active at a time. Suppose that an active exception E_a is present during the execution of a Java method `f○○`. When E_a was caught, the exception handler in `f○○` wraps it in a new exception E_m and throws E_m to indicate its failure. By so doing, E_m becomes an active exception with the root cause E_a . Before `f○○` returns, remaining executed operations (e.g., code in the `finally` clause) may fail either and raises exceptions. These are called *subsequent exceptions* with respect to the active exception E_m . Unless handled, a subsequent exception overrides the original active exception and valuable exception information is lost.

2.2 Views on Exception Handling

Four views on exceptions are relevant to exception handling: the *usage view*, which separates a real exception from a false alarm; the *design view*, which is taken by the developer in treating exceptions; the *tool support view*, which is affected by the exception handling support of chosen programming languages or tools to verify the use of an exception before execution; and the *handling view*, which deals with the structures and dynamics of exception handling. Because exceptions have a variety of meanings in the context of software development, understanding the four views can help the developer better deal with exceptions.

2.2.1 Usage View

Exceptional conditions fall into three categories: *failure*, *result classification* (e.g., representing an end of file event), and *monitoring* (e.g., measuring computational progress) [8]. Exceptions in the latter two categories are actually not exceptional at all; they denote *notifications*. Although it can be argued that exceptions should only be used

to represent atypical conditions and handling should be restricted to the treatments of such exceptions, understanding the difference between failure and notification is important for two reasons. First, conventionally, the usage of exceptions as notifications is not uncommon. The `EOFException` and `InterruptedException` in Java are two examples which lead to closing the file under processing and to terminating the thread under execution, respectively. Second, it is possible for an exception to be originally designed for representing an atypical condition but to be interpreted as a notification in particular application contexts. For example, in the scenario that a DVD player runs a self testing to benchmark the running environment, a continuous loss of frames is regarded as an exception and is reported to the user. However, in playing a movie, the same DVD player directly drops video frames when the available computing power is not sufficient to decode or to draw all frames. In the latter case, if an exception was reported to the user for every lost frame, the application would end up being unusable due to the messes of exceptions. In this context, a frame loss exception is best considered as a notification.

2.2.2 Design View

In the design view, an exception is *declared* if it is part of a component's interface specification that is explicitly visible to the component's caller. Otherwise, an exception is *undeclared*. One way to declare an exception is to place it in a component's signature written in source code or in an interface definition language. Another way to declare an exception is to write it in the document of application programming interfaces (API). A declared exception marks an *anticipated* or *expected* exceptional condition that the caller is expected to deal with at runtime. Thus, declared exceptions are used to represent errors and failures rooted in component faults. In contrast, an undeclared exception represents an *unanticipated* or *unexpected* error condition which the caller

cannot be expected to repair at runtime. An undeclared exception originates from design faults. Although it is possible to handle design faults with exception handling mechanisms, doing so shifts the focus from exception handling to *fault-tolerant programming* [9]. In exception handling, design faults are assumed to be reported by programs at runtime and fixed by developers at maintenance time. This practice is suitable for general business applications where failures caused by design faults are accepted as a reasonable tradeoff for overall development cost. It is also consistent with common software development practice that design faults should be eliminated during the software construction and testing phases rather than tolerated during runtime. Without separating design faults from component faults, the complexity of the exception handling code and the overall system would be greatly increased.

2.2.3 Tool Support View

In the tool support view, an exception is *checked* or *unchecked*. The usage of checked exceptions is verified by software such as compilers or source analysis tools. For example, in the Java language, compilation time processing of a checked exception is governed by the *handle-or-declare* rule [10]. A call to a method that declares to throw a checked exception must be enclosed in a `try` block with the exception caught by an associated `catch` clause, or else the caller must declare to throw the same checked exception (or one of its ancestors) in its method signature. Thus, a checked exception is, by default, a declared exception in the design view. Note that, however, Java allows a method to declare unchecked exceptions in its interface. Thus an unchecked exception can also be a declared exception.

However, the need to declare exceptions in the signature and the way to check their usage in Java unavoidably yield the *interface evolution problem* [4][11]. That is, a change to the exception causes a change to a method's interface, which might cause a

ripple effect of interface changes along the reverse direction of the call chain. In C++, exceptions declared in interfaces are not checked at compile time but at runtime. Thus, C++ avoids the interface evolution problem at compile time. However, programs may encounter unexpected exceptions when violations of exception specifications are detected at runtime.

2.2.4 Handling View

The handling view describes roles, responsibilities and collaborations among the language constructs involved in exception handling. Depending on the programming languages, syntactical constructs are available for implementing these roles. For example, exception handling is conducted with `try` blocks in Java, which contain `try`, `catch`, and `finally` clauses. Normal program logic is placed in the `try` clause; the `catch` clause defines an exception handler based on the exception type to be caught; and the `finally` clause performs cleanup. The handling view could be slightly different from language to language. For example, in C++ there is no corresponding `finally` clauses for cleanup. Resources in C++ are released in destructors. As another example, an exception handler in Eiffel is attached at the method level and all exceptions are caught by one handler. Despite the apparent differences in the use of language constructs, the underlying principles are similar. Understanding the responsibilities of exception handling constructs is vital to writing programs that are both robust and easy to read.

2.3 Exception Handling in Java

Java's exception handling is derived from C++ with some enhancements. All exceptions in Java are data objects which must be subclasses of the `Throwable` class. Java defines three categories of exceptions, namely, `Exception`,

`RuntimeException`, and `Error`. Exceptions in the first category are called *checked exceptions* and in the latter two *unchecked exceptions*. The Java compiler treats checked exceptions differently from the unchecked exceptions. When a checked exception is thrown inside a method, it has to be either caught locally or declared in the method's signature.

Designers of the Java language clearly distinguish the semantics of the three exception categories [1][12]. In short, checked exceptions, such as `IOException` and `SQLException` that are potentially *recoverable*, are descendants of `Exception`. Subclasses of `RuntimeException` (e.g., `NullPointerException`) represent unchecked exceptions that could be produced by any statement. In general, such an unchecked exception is considered to be *unrecoverable*; its intended use is to indicate a *bug* in programs that should be fixed at design or maintenance time. That is, unchecked exception should not be caught and recovered by programmed logic at runtime. Lastly, exceptions that are subclasses of `Error` such as `StackOverflowError` and `OutOfMemoryError` represent disasters that occur within the underlying Java Virtual Machine (JVM) and operating system. Such exceptions should generally not be handled.

Similar to C++, exceptions are signaled with a `throw` statement. Exception handlers are defined in any number of `catch` clauses that immediately follow a `try` block. Lastly, Java allows an optional `finally` clause to be attached to a `try` block for cleanup activities (e.g., releasing resources.) Syntactically, a `try` block consists of three constructs: a `try` clause, zero or more `catch` clauses, and an optional `finally` clause. In addition, at least one `catch` or `finally` clause must exist. If a `finally` clause is present, it is always executed regardless of whether an exceptions is raised or not. The following shows an example of a Java `try` block:

```

try {

    /* protected code block which is expected to signal exceptions */

} catch (IOException e) { /* handler 1 */ }

} catch (SQLException e) { /* handler 2 */ }

} finally () { /* cleanup */ }

```

The selection of exception handlers is matched by the type of the exception caught. A handler catching a generic type of an exception also catches all its subclasses. Thus, the order of `catch` clauses is significant. It is a rule that exception handlers are organized so that exceptions of more specific types are caught before that of more generic types. Java does not have any special construct to define a default handler such as that in Ada and C++. However, it is possible to catch all exceptions not handled by other `catch` clauses with a trailing `catch (Throwable e)` clause.

If no suitable local handler is found, the exception is propagated along the call chain until one is found or top of the call chain is reached, in which case the *thread* executing the method is terminated. A thread can define an `uncaughtException` method to catch all uncaught exceptions and thus prevent the thread from unexpected termination. If an exception is thrown in a nested `try` block and no handler is found, the exception is propagated to the enclosing `try` block, and so forth. Java does not have a *function try block* such as that in C++ [13].

Java support exception specification by allowing a method to declare the exceptions it can throw in its signature. A checked exception must be declared in the signature; otherwise it cannot be thrown or propagated. Although Java allows the declaration of unchecked exceptions, an unchecked exception can be thrown or propagated without declaration. Exceptions are declared with the `throws` keyword, for example:

void login(User u, string pwd) **throws** BadPasswordException;

A method without a `throws` clause indicates that the method does not throw any checked exception. Violations of exception specifications are verified by the compiler and compile-time errors are generated.

Java supports the termination model of exceptions. There is no re-throw mechanism as that in Ada and C++. To throw the caught exception again, the `throw` keyword is used with the caught exception instance as its parameter.

Although a method can declare to throw multiple types of exceptions, in Java and most other programming languages, only one instance of the exceptions is allowed to be signaled to its caller at a time. Thus, if a subsequent exception is thrown, it overwrites the active exception and becomes a new active exception. Consequently, only the latest raised subsequent exception is propagated to the caller. This means that some exception information is lost and the caller cannot describe the complete failure situations.

Chapter 3 A STAGED ROBUSTNESS MODEL

3.1 Goals of Exception Handling

It is generally agreed that exception handling design is a complex problem. Following the general practice of solving a complex problem with divide and conquer, we believe that exception handling design problems are best tackled by conducting a series of small design activities. To this end, we identify four staged exception handling goals for guiding an exception handling design by dividing it into clearly defined intermediate milestones that are easily exercised and verified.

By surveying the literature, we have identified three different levels of robustness that are achieved by commonly adopted exception handling strategies; see Table 3.1. The three robustness levels, referred to as G1, G2 and G3 in Table 3.1, will serve as the goals for directing exception handling design. An additional level, referred to as G0 in Table 3.1, is added to account for the status before exception handling design is reviewed.

As shown in Table 3.1, the level of robustness of components (methods, objects, or software components like JavaBeans) after encountering an exception is described by six elements: name, program's capability to deliver the requested service, the state of the program after handling the exception, how application's lifetime is affected by the exception, known strategies to achieve the robustness level (RL), and also known as. The applicability for the level of robustness is shown in Table 3.2.

Table 3.1: Robustness levels of a component and its effect on the program after encountering an exception

Element	RL G0	RL G1	RL G2	RL G3
name	undefined	error-reporting	state-recovery	behavior-recovery
service	failing implicitly or explicitly	failing explicitly	failing explicitly	delivered
state	unknown or incorrect	unknown or incorrect	correct	correct
lifetime	terminated or continued	terminated	continued	continued
how-achieved	NA	(1) propagating all unhandled exceptions, and (2) catching and reporting them in the main program	(1) error recovery and (2) cleanup	(1) retry, and/or (2) design diversity, data diversity, and functional diversity
also known as	NA	failing-fast	weakly tolerant and organized panic	strongly tolerant, self-repair, self-healing, resilience, and retry

Table 3.2: Applicability for the level of robustness

RL	Applicability
G1	<ul style="list-style-type: none"> ● In the early stage of system development (focusing on the development of normal behavior). ● Prototyping. ● Applying an evolutionary development methodology (e.g., agile methods). ● Time-to-market.
G2	<ul style="list-style-type: none"> ● Outsourcing. ● Designing utility components used in different application domains. ● Behavior-recovery actions should be administered by the user.
G3	<ul style="list-style-type: none"> ● Developing mission critical systems. ● Designing components which have sufficient application context to recover from behavioral failures, e.g., application controllers. ● Implementing connectors in a component-based development environment. ● Behavior-recovery actions are inappropriate to be administered by the user.

Robustness level G0 is given the name “undefined”. Labeling a component G0 declares it the ground zero: you know an exception has been raised; the component has handled it in some undefined way; and the application is in error and may or may not terminate. As will be shown in Section 4.2, there are many ways programmers can be led into creating components of robustness level at G0.

Robustness level G1 is code-named “error-reporting”. For a component internal to a system, reporting an error means throwing an exception; for a component at the system boundary, the exception may be transcribed into an equivalent message intended for the external receiver such as a human operator or an external system. At this level, all exceptions are treated as bugs. After handling the exception, lifetime of the application is terminated with failure notifications sent to the service requester. Prior to terminating, the application is left in an unknown state. G1 is a legitimate goal during

coding and testing because it aims at revealing bugs and forces the developers to deal with the exception before software release. However, a delivered program achieving G1 is likely to be viewed by the end user as having poor quality since the program fails to deliver the request service and terminates in failure. The effort in achieving G1 is small. G1 is also known as failing-fast [14].

Robustness level G2 is code-named “state-recovery”. That is, the faulting component maintains a correct state to continue running and propagates an exception to indicate its failure. By achieving G2, the application gives its user an opportunity to manually correct the faulting condition and retry the failed request. Common methods to implement G2 include error recovery and resource cleanups [15]. Obviously, achieving G2 involves more coding effort and more runtime resources. For example, checkpointing, a common method for error recovery, involves taking a snapshot of the faulting application to capture its entire state and restoring the application with the snapshot when exception occurs [16]. Cleanup is supported in Java and C# through writing resource disposal code in the `finally` clause of a `try` block. G2 is also known as *weakly tolerant* [17] and *organized panic* [18].

Robustness level G3 is code-named “behavior-recovery”. After the faulting component finishes handling the exception, the application delivers the requested service and continues to run normally. Retry, design diversity, data diversity, and functional diversity are common methods to implement G3 [15][16][18]. As can be expected, the coding effort is even greater in order to achieve G3. First, G2 properties must be guaranteed. Then, instead of appealing to the user to resolve the faulting condition, the program retires the failed request automatically. G3 is also known as *strongly tolerant*, *self-repair*, *self-healing*, *resilience*, and *retry* [15][18].

Note that G2 and G3 constitute *failure atomicity* and *designing for recovery*

[17][19], which say that a component should achieve the all-or-nothing property when it encounters an exception.

As can be seen, the exception handling goals are incremental and inclusive: incremental because the robustness level increases as we move up to a higher goal; and inclusive because a higher goal encompasses all the practices of the lower goals. These two properties enable exception handling design to be conducted in a staged manner. Also, note that a component design achieving G3 may fail to achieve G3 but does so with graceful degradation. It can actually reach only G2 (e.g., all retries unsuccessful) or even G1 (e.g., all retries unsuccessful and new exceptions encountered in state restoration or cleanup). See Figure 3.1.

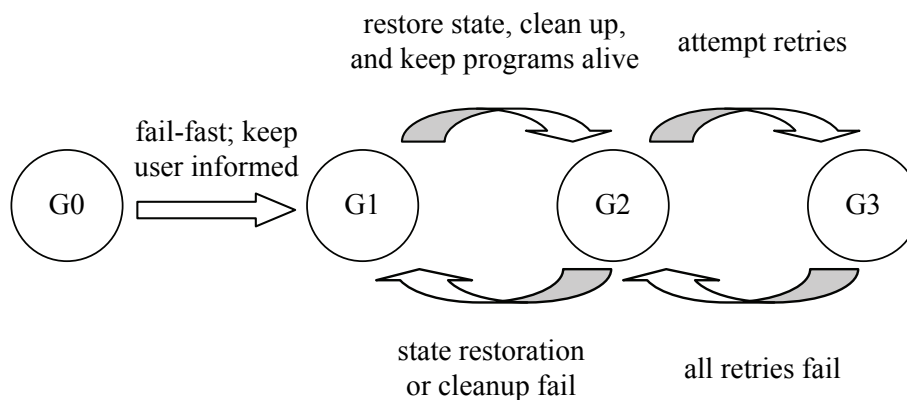


Figure 3.1: Upgrading and degrading exception handling goals

3.2 Checked and Unchecked Exceptions

If the Java language is used, attention must be paid to the distinction of checked and unchecked exceptions, which introduces additional complications. In standard use, an unchecked exception indicates a bug. Normally, you implement failing-fast to achieve G1 by doing nothing about it and allowing your program to abort. In contrast, a checked exception represents an error that your program should deal with. The Java

compiler makes sure you do this by enforcing the handle-or-declare rule: you must either catch the checked exception, or declare it on the caller's interface [10]. While this seems a sound principle for designing robust programs in theory, it is easy for programmers to mishandle checked exceptions in practice and leads to programs that are less robust. For instance, one widely acknowledged way for programmers to mishandle a checked exception is to *ignore* it, which means catching an exception and doing nothing about it [20]. When this happens, the component fails to achieve G1, which makes further exception handling and debugging difficult. For this reason, unchecked exceptions are preferred in several well-known open source projects written in Java, including the Eclipse SWT project (<http://www.eclipse.org>) and the Spring framework (<http://www.springframework.org>).

In a nutshell, wrapping a checked exception into an unchecked exception constitutes a legitimate exception handling strategy that achieves G1. Checked exceptions are best used when the goal is to achieve G2 or G3 because the compiler will remind you about an uncaught checked exception. With no help from the compiler, achieving G2 or G3 with unchecked exceptions requires additional attention by the programmer.

3.2.1 Programming to an interface regarding exceptions

The ripple effect caused by exception induced interface changes is generally regarded as a bad side effect [4][11]. However, this side effect could increase robustness because it reveals the need to modify all effected components to cope with the exception changes [8]. In this regard, *not only checked exceptions but also unchecked exceptions produce interface evaluation except that it's explicit in the former and implicit in the*

latter.

Being an integrated part of the signature, checked exceptions extends the fundamental object-oriented principle *programming to an interface* to cover exceptions [21]. In other words, checked exceptions provides *abstract coupling* between a component and its callers concerning errors [22]. Errors become predictable which enable the caller to practice designing for recovery. For this reason, checked exceptions declared in interfaces should represent errors that are meaningful in the problem domain. Because an application can have many different abstractions layers, aggregating exceptions in lower-level abstractions to higher-level abstractions will inevitably cause the *scalability problem* [4], which is a misuse of checked exceptions.

Regardless of the scalability problem, if an interface declares several checked exceptions, this might be a bad smell indicating the violation of the *single-responsibility principle* [23]. It is a sign to separate different responsibilities from the interface.

When unchecked exceptions are used, these object-oriented principles should be followed as well. Otherwise, exceptions become merely an error reporting channel and designing for recovery is difficult. This is why all exceptions in C# are by definition *unrecoverable* [24] and local exception handlers are either absent or simply dumping the exceptions to the console. There is little you can do about unanticipated runtime errors. Therefore, to support designing for recovery with unchecked exceptions, you should design two semantically clear exception archetypes: recoverable and unrecoverable. The exception hierarchy design in the Eclipse Standard Widget Toolkit project gives such an example where `SWTException` and `SWTError` represent recoverable and unrecoverable errors, respectively [25].

Nevertheless, changing declared checked exceptions indeed causes the change of the interface which must be dealt with. In fact, this is a common phenomenon of static

typing languages. Interface evolution is induced per part of the method signature changes, including the return type, the method name, parameters, and exceptions. This is actually a *component versioning problem* [26]. Thus, the solution to the interface evolution problem is the same as that in the component versioning problem:

- distinguishing a public interface from a published interface [27],
- freely refactoring unpublished interfaces if necessary,
- declaring published interfaces with circumspection, and
- making published interfaces immutable [26].

3.3 Exception Handling Primitives and Costs

As we go up in the robustness level, the exception handling processing becomes more complicated. Table 3.3 lists the primitive operations that are used in exception handling.

Note that the exception handling primitive operations themselves can fail, and we need a view on how to deal with such exceptions. In the proposed model, G1 primitives are directly supported by programming languages and are not supposed to fail. For example, exception signaling and propagation should rarely fail. However, when they do fail, an unchecked exception is generally raised by the execution environment to indicate a fatal error. Under such circumstances, the unchecked exception is allowed to overwrite the original exception. Primitives operations at G2 and G3, on the other hand, are not unlike regular code for normal program logic. When they fail, only an exception declared in the failing method's interface should be thrown. Any exception raised by executing the primitive operations should be recorded in the declared exception.

Table 3.3: Exception handling primitive operations

RL	Primitive	Meaning
G1	detecting	Detecting an erroneous state of a component with program logic such as assertions, checks, and acceptance tests.
	signaling	Signaling or throwing an exception to indicate an error or a failure.
	catching	Catching an exception to start the handling process. Because catching checked exceptions and unchecked exceptions require different effort, they are distinguished and denoted by catching_c and catching_u respectively.
	wrapping	Transforming an exception into another type. This can imply a change of exception type.
	declaring	Declaring and/or documenting an exception to be thrown by a component. Similar to catching, declaring checked exceptions and unchecked exceptions are distinguished and denoted by declaring_c and declaring_u respectively. If the exception is designed up-front, it is denoted by $\text{declaring}_{c, \text{pre}}$ and $\text{declaring}_{u, \text{pre}}$.
	informing	Informing the users or developers of the occurrence of the exception. Possible actions include popping up an error dialog and logging.

Table 3.3: Exception handling primitive operations (cont.)

RL	Primitive	Meaning
G2	error-handling	Recovering the system from an erroneous state. Common methods include backward recovery, forward recovery, and compensation.
	cleanup	Returning resources to the runtime environment. A cleanup operation is performed regardless of the occurrence of an exception.
	interface-changing	Changing the interface of a component due to declaring.
G3	fault-elimination	Removing the exceptional condition and preventing the fault from being activated again.
	alternative	Designing and implementing alternatives. Common approaches include design diversity, data diversity, and functional diversity.
	selection mechanism	Designing a mechanism for selecting suitable alternatives to replace the failed one regarding the raised exception. The selection mechanism may employ static, dynamic, or rule-based strategies.

Table 3.4 spells out the primitive operations that are performed in reaching a robustness level with a cost equation. In particular, the exception handling cost of a non-top-level component at G1 is optionally (1) raising a new exception (cost of signaling); (2) encountering an unchecked exception (no cost since it is automatically propagated); or (3) encountering a checked exception and wrapping it to a predefined

unchecked `UnhandledException` so that the wrapped one can be implicitly propagated subsequently (cost of wrapping and signaling). The wrapped exception also acts as a marker or an indicator. This way you know how many *exception holes* there are in the system and where to find them if needed.

Table 3.4: Exception handling cost analysis

RL	Cost Equation
G1	<p>For try blocks not at the top level:</p> $cost(error-reporting) = cost(declaring) +$ <div style="display: flex; align-items: center; justify-content: center;"> <div style="font-size: 3em; margin-right: 10px;">{</div> <div style="text-align: left;"> $cost(detecting) + cost(signaling)$ new exceptions 0 unchecked exceptions $cost(catching) + cost(wrapping) + cost(signaling)$ checked exceptions </div> </div> <p>For the top-level try block:</p> $cost(error-reporting) = cost(informing)$
G2	$cost(state-recovery) = cost(error-reporting) + cost(error-handling) + cost(cleanup) + cost(interface-changing)$
G3	$cost(behavior-recovery) = cost(state-recovery) + cost(fault-elimination) + cost(alternative) + cost(selection\ mechanism)$

The top-level G1 component is a *safety net* [28]. It catches all types of uncaught exceptions and therefore prevents the system from unexpected termination and notifies users or developers about the occurrence of exceptions (cost of informing). All of the

above handling operations in G1 are supported by most mainstream programming languages and can be performed fairly mechanically.

Due to the inclusive nature of the robustness model, the exception handling cost of a higher goal includes the cost of a lower robustness level. The cost of implementing G2 and G3 exception handling can be derived from Table 3.3 and are not discussed further.

It should be noticed that Table 3.3 is a description of one essential set of primitive operations involved in the implementation of each robustness level; it is not a prescription. You are free to add new operations to any of the robustness levels when needed. Also note that we do not assign any numerical value to the cost equations in Table 3.4 except the zero in G1 because the actual cost of the primitive operations, essentially those in G2 and G3, depends on the implementation details. For example, the cost of error-handling is relatively low in database applications where transaction is supported via JDBC API. In contrast, programmed compensation is required in manipulating file systems in which transaction is usually not available. Thus, the cost equations act as templates where the parameters are given by developers. However, even if the detailed design of the primitive operations is yet devised, the cost equations are also valuable because the cost involved in exception handling becomes explicit. In our experience, customers and the management tend to implicitly think that all components in the delivered application ought to be G3; there is no reason to buy or delivery a fragile application. However, due to project attributes such as time-to-market and limited budgets, G3 components are rare in practice. When the exception handling cost becomes explicit, educated tradeoffs can be made with respect to project attributes such as time-to-market, budgets, iteration plans, robustness, application domains, and so on.

3.4 Documenting Exception Handling Goals

We advocate documenting a component's robustness levels (i.e., exception handling goals) in source code. In particular, such meta-information is recorded with Java's *annotations* and C#'s *custom attributes* [1][24] (or in comment if a language does not have an annotating mechanism). Figure 3.2(a) and Figure 3.2(b) shows a `writeFile` program written in Java and C#, respectively. Table 3.5 illustrates three different robustness levels of the `writeFile` method in Java and C#. The designed annotations and custom attributes are shown in Figure 3.3(a) and Figure 3.3(b), respectively. Table 3.5 spells out, for example, that when `writeFile` is G2 and a `WriteFileException` is thrown, `writeFile` ensures the correctness of its state; for all other exceptions, only error-reporting capability (i.e., G1) is guaranteed.

```
01 public void writeFile(String filePrefix, String data) throws IOException {  
02     Writer writer = null;  
03     try {  
04         writer = new FileWriter(filePrefix+"_"+data+".txt");  
05         writer.write(data);  
06     } finally { // code for cleanup }  
07 }
```

(a)

```
01 public void writeFile(String filePrefix, String data) {  
02     TextWriter writer = null;  
03     try {  
04         writer = new StreamWriter(filePrefix+"_"+data+".txt");  
05         writer.Write(data);  
06     } finally { // code for cleanup }  
07 }
```

(b)

Figure 3.2: Example programs written in (a) Java and (b) C#

Table 3.5: Robustness level of the `writeFile` method documented with Java annotations and with C# custom attributes

RL	Annotated robustness levels	
G1	Java	<code>@Robustness(@RL(level=1, exception=Throwable.class))</code>
	C#	<code>[Robustness(1, typeof(Exception))]</code>
G2	Java	<code>@Robustness({ @RL(level=2, exception=WriteFileException.class), @RL(level=1, exception=Throwable.class)})</code>
	C#	<code>[Robustness(2, typeof(WriteFileException))]</code> <code>[Robustness(1, typeof(Exception))]</code>
G3	Java	<code>@Robustness({ @RL(level=3, exception=WriteFileException.class), @RL(level=1, exception=Throwable.class)})</code>
	C#	<code>[Robustness(3, typeof(WriteFileException))]</code> <code>[Robustness(1, typeof(Exception))]</code>

Note that the annotation does not need a field for documenting checked or unchecked exceptions because such information can be derived from the exception type. Also note that when `writeFile` is upgraded to G2, a new checked exception `WriteFileException` is declared which is thrown at the exception handler of `IOException` to explicitly represent a failure.

```
// Robustness.java

public @interface Robustness {

    RL [] value();

}

// RL.java

public @interface RL {

    int level() default 1;

    Class exception() default Throwable.class;

}
```

(a)

```
// Robustness.cs

public class Robustness : System.Attribute

    private int level;

    private Type exception;

    public Robustness(int level, Type exception) {

        this.level = level;

        this.exception = exception;

    }

    public int Level{ get { return level; } }

    public Type Exception {

        get { return exception; }

    }

}
```

Figure 3.3: The source code of robustness documentation using (a) Java’s annotations, and (b) C#’s custom attributes

3.5 Consequences

The proposed robustness model has several important consequences.

- *Robustness without costly up-front design.* Exception handling cost is amortized through software development iterations toward desired robustness levels.
- *Freedom in design without chaos.* The ability to evolve component robustness provides developers a flexible way to handle exceptions according to different factors, such as application contexts, development methodologies, software architectures, component characteristics, and resource constraints. In addition, the flexibility nature of unchecked exceptions is remained while supporting “programming to specification” regarding exceptions with the robustness annotations. For checked exceptions, G1 eliminates swallowing of exceptions, which is one of the most serious problems of the misuse of checked exceptions.
- *Facilitating tool support.* The robustness annotations enable the development of exception handling supporting tools, including compiler-time checking for unchecked exceptions, automatic code generation of exception handling, and automatic exception documentation. See Chapter 6.3 and 6.4 for more information.
- *Explicit interface evaluation.* G1 prevents immature interface design revealing implementation detail with checked exceptions, which causes uncontrolled ripple effect when the exception changes. With tool support, the robustness annotations make the effect of interface evaluation explicit with unchecked exceptions.
- *Guiding exception handling implementation.* Once a robustness level is determined, its corresponding implementation is clear.
- *Independent of languages.* The model is applicable for languages supporting checked and/or unchecked exceptions.

Chapter 4 EXCEPTION HANDLING REFACTORING

In Chapter 3, we have established a robustness model with four exception handling goals to support exception handling design. However, like most software design problems, even if developers are requested to design with exception handling beforehand, it is very difficult to get the right design at the first shot. Therefore, improving exception handling design after software is constructed is necessary. In this chapter, we apply refactoring to incrementally improve exception handling design. We introduce exception handling smells that hinder the achievement of the exception handling goals and propose exception handling refactorings to eliminate the smells.

4.1 Refactoring

Refactoring is a commonly exercised practice in many software development methods nowadays, most notably in agile methods. Since refactoring aims at “improving the internal structure of a software system without altering its external behavior” [29], it is regarded as a disciplined way to enhance the system’s quality attributes (i.e., non-functional requirements) such as understandability, testability, modifiability, and so on. Ideally, such quality attributes should be designed up front instead of being improved after the system is constructed. However, in development where requirements are only poorly understood and can change often, it is extremely difficult, if not impossible, to get to the right design before coding starts. Therefore, refactoring is widely accepted as a necessary rework that balances the needs to respond to uncertainty and to achieve a good design.

To perform refactoring, three general steps are involved: (1) identifying code

smells that degrade the design, (2) applying refactorings to remove the code smells, and (3) verifying satisfaction with the refactored program. According to the extent of change to the source code, refactorings can be divided into two categories [29]: (1) *small refactorings* or *primitive refactorings*, which introduce individual changes such as renaming a method or relocating a method, and (2) *big refactorings* or *composite refactorings*, which apply a coherent series of small refactorings to achieve a larger design goal such as introducing a design pattern and untangling an inheritance mess. Applying small refactorings is relatively easy because the change is small, the scope is clear, and thus the result is immediately verifiable. In contrast, conducting big refactorings is much more difficult since instant verification on achieving the goal may not be possible.

In this research, we present *exception handling refactorings* (hereafter referred to as *EH refactorings*) which improve the software design pertaining to abnormal or exceptional behavior. By definition, refactoring should not change the behavior of the software. In practice, however, it is difficult to precisely define behavior and to effectively verify whether behavior remains unchanged [30]. From an exception handling perspective, a system's behavior can be categorized into the normal part and the abnormal part [6]. Most existing refactorings focus on improving the software design pertaining to the normal behavior of a system. In this regard, refactoring is only requested not to alter the external normal behavior. In contrast, EH refactorings enhance the system's robustness by possibly changing the exceptional behavior but without altering the system's normal behavior.

Applying refactorings to improve the abnormal behavior is a relatively new idea and it can be argued that EH refactorings differ from normal refactorings regarding two key issues. First, code smells in exception handling not only degrade the design quality

but also tend to cause bugs in code. Second, since EH refactorings remove code smells, they can fix the smell-caused bugs and therefore improve robustness by changing the exceptional behavior of the refactored program.

4.2 Bad Smells

Deviation from the practices required to achieve the exception handling goals compromises the robustness of the program under development. The deviation manifests as *code smells* in [29] and should be avoided in exception handling code as well. In this dissertation, we identify six exception handling code smells, hereafter referred to as *EH smells*.

4.2.1 Return code

Many programming languages – most notably, the C language – use return code to indicate error conditions of the execution of a function. However, using return code as an error reporting mechanism has one major drawback: normal code and error handling code is mixed and cluttered. If each method invocation is checked for all possible error conditions, code becomes unintelligible and difficult to maintain.

Moreover, mixing the use of return code and exceptions further exacerbates the problem of code tangle. The situation can easily occur when writing applications in exception-supported programming languages (e.g., Java and C#) that make calls to APIs of legacy systems implemented in C. Programs are confusing and difficult to maintain when exceptions and return code are arbitrarily mixed.

4.2.2 Ignored checked exception

Ignored checked exceptions reduce robustness and make debugging difficult. A checked exception indicates the occurrence of an expected and recoverable error

[10][12]. If a checked exception is caught but nothing is done to deal with it, the program is pretending that all is fine when in fact something is wrong. This can ultimately lead to program failures and leave no clues as to what may have caused the failures.

4.2.3 Unprotected main program

Uncaught exceptions that are spilt from application/thread boundaries to the execution environment will eventually terminate the application/thread in an unexpected way. When users encounter such an unexpected termination, the application is usually regarded as having poor quality.

4.2.4 Dummy handler

A popular way to handle exceptions is to write code like:

```
catch (SomeException e) { e.printStackTrace(); }, or
```

```
catch (SomeException e) { System.out.println(e.toString()); }
```

While this seems slightly better than ignoring the exception since exception information is displayed or logged, nothing is done to fix the program state and allowing the execution to continue can lead to program failure. This gives the name of the smell: dummy handler. Dummy handlers create a false impression that the exception has been properly handled. Like ignored checked exceptions, they put the program's robustness at risk.

4.2.5 Nested try block

Unconstrained use of nested code constructs such as conditional (e.g., if-then-else), loop (e.g., for and while loops) produces programs with complex

structures that are difficult to read, test, and maintain. The same can be said about nested `try` blocks.

4.2.6 Catch clause as spare handler

If an associated catch clause provides an alternative implementation of the actions attempted in the `try` clause, it effectively becomes a spare handler of the `try` clause. Since the alternative implementation can encounter exceptions as well, a spare handler can easily lead to nested `try` blocks.

4.3 Refactorings

The seven refactorings of Table 4.1 are applied to remove the EH smells and to achieve the proposed exception handling goals in Chapter 3.1. The first one, *Replace Error Code with Exception*, is a well-known refactoring presented in [29] and is not discussed further. In this section, the six remaining refactorings are presented in detail. Note that while these refactorings (or parts thereof) appeared in alternative forms in the literature, the refactorings in their present form were extracted from our experiences in applying and refining them in more than ten projects in the banking and software engineering domains during the time span from spring 2005 to summer 2008. In Chapter 6.2, one such case is studied in detail.

Martin Fowler's format is used in presenting the refactorings [29], which consists of five parts: name, summary, motivation, mechanics, and examples.

Table 4.1: Exception handling smells, refactorings, and goals

EH smell	Refactoring	Achieving RL
Return code	Replace Error Code with Exception	G1
Ignored checked exception	Replace Ignored Checked Exception with Unchecked Exception	G1
Unprotected main program	Avoid Unexpected Termination with Big Outer Try Block	G1
Dummy handler	Replace Dummy Handler with Rethrow	G1
Nested try block	Replace Nested Try Block with Method	G2
Ignored checked exception, Dummy handler	Introduce Checkpoint Class	G2
Spare handler	Introduce Resourceful Try Clause	G3

4.3.1 Replace ignored checked exception with unchecked exception

You have caught a checked exception but done nothing to deal with it.

Wrap the caught checked exception with a predefined unchecked `UnhandledException` and throw the unchecked exception.

```
public void writeFile(String fileName, String data) {
    Writer writer = null;
    try {
```

```

        writer = new FileWriter(fileName);  /* may throw an IOException */

        writer.write(data);  /* may throw an IOException */

    }

    catch (IOException e) {

        /* ignoring the exception */

    }

    finally {  /* code for cleanup */  }

}

```

↓

```

public void writeFile(String fileName, String data) {

    Writer writer = null;

    try {

        writer = new FileWriter(fileName);  /* may throw an IOException */

        writer.write(data);  /* may throw an IOException */

    }

    catch (IOException e) {

        throw new UnhandledException(e, "message");

    }

    finally {  /* code for cleanup */  }

}

```

4.3.1.1 Motivation

When you make a call to a method that declares to throw a checked exception, you are bound by the Java compiler to either handle the checked exception or to declare the same checked exception in your method's interface. Since taking the latter option bloats the caller's interface and leads to the problem of interface evolution [4][11], it is more

sensible to handle the checked exception.

However, deciding on how to handle the checked exception can be a challenging design issue in itself. Or, it may be that you just want to concentrate on coding up the normal behavior for the time being and come back to deal with the exception later. In either case, it is easy for you to end up with handling code like this:

```
catch (IOException e) { /* TO DO */ }
```

The comment signals your intention to come back to deal with the exception. The comment, unfortunately, has no binding power and you may never actually do anything about it. In effect, you have ignored the checked exception and your program's robustness is in jeopardy. Specifically, your program fails to achieve G1 since the error is ignored rather than reported.

Therefore, instead of catching a checked exception and doing nothing about it, wrap the checked exception into an unchecked `UnhandledException` and throw the latter. There are two consequences by so doing. First, since the wrapped unchecked exception automatically propagates through the call chain, you are no longer bound by the handle-or-declare rule so far as the original checked exception is concerned. This gives you the freedom to concentrate on the normal behavior of your program. Second, the cause of the original checked exception is preserved and can be extracted by applying *Avoid Unexpected Termination with Big Outer Try Block* in the outer-most component of your program. The information can be useful in debugging.

4.3.1.2 Mechanics

- i. Define an unchecked `UnhandledException` class as to represent the situation that a checked exception is not handled.
- ii. Create a new instance of `UnhandledException`.

- iii. Chain the ignored exception to the instance of `UnhandledException`.
- iv. Add to the exception a string with a suitable message to document the exceptional situation.
- v. Throw the instance of `UnhandledException`.
- vi. Compile and test.

4.3.1.3 Example

You have a method `WriteFile` which writes a string to a file. Your top priority is to implement the functionality of the method. You use Java's file manipulation operations that declare to throw `IOException`. You catch the `IOException` to comply with Java's handle-or-declare rule. You do not devise any exception handling code and leave the catch clause empty; see the pre-refactored code fragment shown above.

You want to report the `IOException` but do not want to declare it because doing so will force callers of `writeFile` to deal with the `IOException` as well. Thus, you throw an instance of `UnhandledException` in the catch clause.

```
catch (IOException e) { throw new UnhandledException(e, "message"); }
```

Since the failure of `writeFile` is reported as an unchecked exception, callers of `writeFile` do not need to deal with it. Thus, the change is local to the `writeFile` method. In contrast, you can alternatively rethrow a checked exception, but this creates a ripple effect since every method along the call chain must be changed to deal with the checked exception.

You then apply *Avoid Unexpected Termination with Big Outer Try Block* to catch the unchecked exception at the main program where useful actions are performed, e.g., logging the exception, showing an error dialog to the user, and gracefully shutting down

the application.

4.3.2 Avoid unexpected termination with big outer try block

An exception is propagated to the outer-most component and terminates your application.

Enclose the outer-most component within a try block which catches all exceptions and displays/logs the exception.

```
static public void main(String[] args) {    /* some code */ }
```

↓

```
static public void main(String[] args) {  
    try {        /* some code */    }  
    catch (Throwable e) {    /* displaying and/or logging the exception */    }  
}
```

4.3.2.1 Motivation

To the operating system, the main program is the entry point of your application. All uncaught exceptions are ultimately propagated to the main program. If they are not caught, the main program aborts abnormally. When this happens, it indicates the presence of a bug. Also, although some error messages are generated by the operating system, they may go unnoticed or are intentionally ignored by the human operators (e.g., developers during testing and users during operational use). G1 is achieved only partially since exception information becomes lost.

Apply *Avoid Unexpected Termination with Big Outer Try Block* to protect your main program from outright failure. By catching all exceptions, you can soften the failure by displaying an appropriate message to the user. Furthermore, with proper

handling actions (e.g., logging,) you avoid the loss of valuable debugging information.

In single-threaded applications, it is easy to locate the main program of the application and to check whether it is unprotected from unexpected termination caused by uncaught exceptions. In multi-threaded applications, protection must be provided by applying *Avoid Unexpected Termination with Big Outer Try Block* to every thread at its entry point should it be the case that exceptions are not propagated across thread boundaries, as in Java.

4.3.2.2 Mechanics

- i. Find the main program of your application. For multi-threaded applications, identify each thread's entry point.
- ii. Enclose the main program and each thread's entry point with a `try` block.
- iii. Write a blanket catch (a `catch` clause that catches all exceptions) for the `try` block.
- iv. Before terminating the main program, display and/or log the caught exception in the body of the `catch` clause.
- v. Compile and test.

4.3.2.3 Example

Your application is terminated unexpectedly due to unhandled exceptions propagated to the runtime environment:

```
static public void main (String[] args) { /* some code */ }
```

You want to put a `try` block in the main program of your application. You begin by finding the main program of your application, write a `try` block with a blanket catch, and move the body of the main program into the `try` clause. In the `catch` clause you

display the caught exception:

```
static public void main(String[] args) {  
    try { /*some code */}  
    catch (Throwable e) { /* display e before termination */}  
}
```

4.3.3 Replace dummy handler with rethrow

You use a dummy handler to handle exceptions.

Remove the dummy handler. Wrap the caught exception to a predefined unchecked

UnhandledException and throw the unchecked exception.

```
catch (AnException e) { e.printStackTrace(); }
```

↓

```
catch (AnException e) { throw new UnhandledException(e, "message"); }
```

4.3.3.1 Motivation

Next to ignoring checked exceptions, dummy handlers are a popular means to satisfy the handle-or-declare rule. Notably, sophisticated coding assistance tools in modern IDEs may unintentionally encourage the use of dummy handlers. For example, Eclipse (<http://www.eclipse.org/>), a popular Java IDE and open development platform, provides a mechanism called “quick fix” to facilitate correcting syntax errors. One of the two quick fix proposals available for checked exceptions is “surround with try/catch.” By accepting the proposal, Eclipse automatically adds a dummy handler to deal with the exception:

```
catch (AnException e) {
```

```
// TODO Auto-generated catch block  
e.printStackTrace();  
}
```

With quick fix, the dummy handlers are literally only two mouse-clicks away (i.e., clicking the quick fix icon and selecting the “surround with try/catch” proposal). Programs with dummy handlers fail to achieve G1.

Therefore, instead of blindly accepting the proposal to generate a dummy handler, throw an unchecked exception to preserve the exception. Apply *Avoid Unexpected Termination with Big Outer Try Block* in the outer-most component of your program to prevent unexpected termination of your application.

Note that dummy handlers in Java `finally` clauses are acceptable, since exceptions thrown inside `finally` clauses mask any previously active exceptions. This is a well-known problem of the Java exception handling mechanism [5]. A pattern-based solution to the problem is proposed in Chapter 5.

4.3.3.2 Mechanics

- i. Remove the code that constitutes the dummy handler.
- ii. Apply *Replace Ignored Checked Exception with Unchecked Exception* refactoring.
- iii. Compile and test.

4.3.3.3 Example

The situation of using *Replace Dummy Handler with Rethrow* is similar to that of using *Replace Ignored Checked Exception with Unchecked Exception*. In the former, you seek for a dummy handler; in the later you seek for an ignored exception. Both of these refactorings throw an unchecked exception:

```
throw new UnhandledException(e, “message”);
```

4.3.4 Replace nested try block with method

You have a nested `try` block.

Extract the nested `try` block to a method.

```
FileInputStream in = null;
```

```
try { in = new FileInputStream(...); }  
  
finally {  
  
    try { if (in != null) in.close (); }  
  
    catch (IOException e) { /* log the exception */ }  
  
}
```

↓

```
FileInputStream in = null;
```

```
try { in = new FileInputStream(...); }  
  
finally { closeIO (in); }
```

```
private void closeIO (Closeable c) {
```

```
    try {  
  
        if (c != null) c.close ();  
  
    catch (IOException e) { /* log the exception */ }
```

```
}
```

4.3.4.1 Motivation

In Java and C#, programmers are free to enclose a nested `try` block in `try`,

`catch`, and `finally` clauses. There are two common reasons behind the use of nested `try` blocks.

- Figure 4.1 shows an example taken from [31]. The method `makeTransfer` updates database records through JDBC APIs. One of the record fields is the invoker's Internet address, which is provided by calling Java's `InetAddress` class (line 7). When this fails, a default value of the invoker's Internet address is provided (line 8).
- Forced by the language. In Java, when statements in the `catch` and `finally` clauses include calls to methods (e.g., state restoration or cleanup) that throw checked exceptions, programmers may incline to enclose these statements with a nested `try` block.

```
01 public void makeTransfer (long AcctNo, float amount) {  
02  
03     try {  
04         /* (1) configure database connection */  
05         ...  
06         String localhost = "";  
07         try { localhost = InetAddress.getLocalHost().toString(); }  
08         catch (UnknownHostException ex) { localhost = "localhost/127.0.0.1"; }  
09         ...  
10         /* (2) update database */  
11         ...  
12     } catch (SQLException e) { ... }  
13 }
```

Figure 4.1: A nested `try` block providing an inline alternative

Each of the use of the nested `try` block yields complicated program structures and

easily results in a *long method* [29].

To avoid deeply nested `try` blocks, apply *Replace Nested Try Block with Method*. If the replaced `try` block implements exception handling logic that can be used in multiple places, extracting it into a method further prevents code duplication. In the pre-refactored code fragment shown in the summary of this refactoring, the `finally` clause implements actions taken to release stream-based resources. Such actions are used in places wherever resource cleanups are required. After extracting the nested `try` block into the `closeIO` method, code in the refactored `finally` clause becomes much clearer.

4.3.4.2 Mechanics

- i. Apply *Extract Method* [29] to each nested `try` block.
- ii. Replace the nested `try` block with a call to the extracted method.
- iii. Compile and test.

4.3.4.3 Example

You have a method `update` that maintains user's data in a database via JDBC APIs. You create a `Connection` object and a `PreparedStatement` object to connect to the database and to manipulate records in the database, respectively. You enclose JDBC operations in a `try` block because they declare to throw `SQLException`. You invoke a `close` method on the `Connection` object and the `PreparedStatement` object to release shared database resources, respectively:

```
public void update() {  
  
    Connection conn = null;      PreparedStatement ps = null;  
  
    try {    conn = getConnection();
```

```

        ps = conn.prepareStatement (/* update user's data */);

        ...}

        catch (SQLException e) {/* rollback */}

    finally {    try {    if (ps != null) ps.close();

                    if (conn != null) conn.close(); }

                catch (Exception e) {/* log exception */}

            }

        }

```

The above `finally` clause demonstrates a common implementation to release database resources, which makes use of nested `try` blocks. However, the implementation contains a resource leak bug: if the invocation of `close` on the `ps` object fails, the execution flow jumps to the `catch` clause and the code to release the `Connection` object is skipped.

You apply *Replace Nested Try Block with Method* to avoid nested `try` blocks and to fix this bug. You create two overloaded `close` methods to release the database resources:

```

public static void close(PreparedStatement obj) {

    try {    if (obj!= null) obj.close();    }

    catch (Exception e) { /* log exception */}

}

public static void close(Connection obj) {

    try {    if (obj!= null) obj.close();    }

```

```

        catch (Exception e) { /* log exception */}

    }

```

You modify the original `finally` clause to invoke the two overloaded `close` method by passing the `PreparedStatement` object and the `Connection` object to them, respectively.

```

finally {    close (ps);    close(conn);    }

```

4.3.5 Introduce checkpoint class

You have a `try` clause that changes the state of the application. In case the `try` clause encounters an exception, you want to make sure that the program remains in a correct state.

Create a checkpoint class for state management, which has methods for state preservation, state restoration, and checkpoint disposal.

```

public void foo () throws FailureException {

    try { /* code that may change the state of the object */}

    catch (AnException e) { throw new FailureException(e);}

    finally { /* code for cleanup */}

}

```

↓

```

public void foo () throws FailureException {

    Checkpoint cp = new Checkpoint (/* parameters */);

    try {

        cp. establish (); /* establish a checkpoint */

```

```

        /* code that may change the state of the object */ }

    catch (AnException e) {

        cp.restore ();  /* restore the checkpoint */

        throw new FailureException(e);    }

    finally { cp.drop();    }

}

```

4.3.5.1 Motivation

The normal execution of a method can change the state of the application. If the application were to achieve G2, i.e., to continue running normally after encountering exceptions, the faulting method must see to it that the application is restored to a correct state. This can be achieved through the use of checkpoints. Three operations are involved: establishing, restoring, and dropping a checkpoint. State information is saved in a checkpoint before state-modifying actions are performed in the `try` clause. If the actions in the `try` clause fail, the checkpoint is used to restore to the previous correct state. The checkpoint is dropped in the `finally` clause.

Encapsulating operations and data into a class is essential in object-oriented design and programming. It is reasonable to encapsulate the three operations of checkpoint manipulation to a checkpoint class. Using a checkpoint class simplifies the management of exception handling code, fosters reusability, and reduces code duplication.

4.3.5.2 Mechanics

- i. Create a checkpoint class with three methods: `establish`, `restore`, and `drop`.
- ii. Implement the three methods of the checkpoint class.
- iii. Compile and test.

- iv. Invoke the `establish` method to produce a checkpoint in the entry of the `try` clause.
- v. Invoke the `restore` method in `catch` clauses to restore to the previous state.
- vi. Invoke the `drop` method in the `finally` clause to abandon the checkpoint.
- vii. Compile and test.

4.3.5.3 Examples

You have a `checkout` method which downloads a number of source files from a concurrent versions system (CVS) repository to a local workspace. To prevent the local workspace from crashing due to network disruption during downloading, you make a checkpoint by taking a snapshot of the local workspace before downloading. The local workspace is recovered from the snapshot if exceptions occur during downloading. The snapshot is dropped before checkout returns:

```
public void checkout(String repository, String workspace, String tmp)

                                throws CheckoutException {

    try {

        makeSnapshot(workspace, tmp);

        download(repository, workspace); /* may throw an IOException */

    catch (IOException e) { restore(tmp, workspace);

        throw new CheckoutException(e); }

    finally { dropSnapshot(tmp); }

}
```

You introduce a checkpoint class that manages the state of files:

```

public FileCheckpoint {

    private String    _workspace = null; private String    _tmp = null;

    public FileCheckpoint(String workspace, String tmp) { /* constructor */};

    public void establish() { /* code for establishing */ };

    public void restore() { /* code for restoring */};

    public void drop() { /* code for dropping */};

}

```

You first create an instance of the `FileCheckpoint` class. Then, you replace invocations of `makeSnapshot`, `restore`, and `dropSnapshot` in the original `checkout` method with invocations of `establish`, `restore`, and `drop` of the `FileCheckpoint` class:

```

public void checkout(String repository, String workspace, String tmp)

                                throws CheckoutException {

    FileCheckpoint fcp = new FileCheckpoint(workspace, tmp);

    try {

        fcp.establish();

        download(repository, workspace);  /* may throw an IOException */

    catch (IOException e) { fcp.restore();

        throw new CheckoutException(e); }

    finally { fcp.drop(); }

}

```

4.3.6 Introduce resourceful try clause

The implementation of a `catch` clause acts as a spare of the computation performed in the corresponding `try` clause. The execution of the spare implementation throws checked exceptions.

Move the spare implementation in the `catch` clause to the `try` clause and re-execute the `try` block when encountering exceptions. Introduce variables to control the retry process and design a selection mechanism to choose between the primary and the alternatives.

```
try { /* primary */ }
catch (SomeException e) {
    try { /* alternative */ }
    catch (AnotherException e) { throw new FailureException(e); }
}
```

↓

```
int attempt = 0;    int maxAttempt = 2;    boolean retry = false;

do {

    try { retry = false;

        if (attempt == 0)    { /* primary */ }

        else                { /* alternative */ }

        catch (SomeException e) {

            attempt++;    retry = true;

            if (attempt > maxAttempt)    throw new FailureException (e);

        }

    }
```

} while (attempt<= maxAttempt && retry)

4.3.6.1 Motivation

Retrying is a widely acknowledged exception handling strategy that is useful for masking failures caused by transient conditions such as system overloading or temporarily unavailable resources [10][18][19]. In a language that supports a retry model of exception handling (e.g., Eiffel), retrying is directly supported [18]. However, in languages like Java and C#, which adopt a termination model for exception handling, implementation of retrying requires extra coding effort. In practice, many programmers simply use nested `try` blocks to provide alternatives; see the `try` block in lines 7 and 8 in , which implements a spare handler for the `try` clause beginning in line 3. Although such use of spare handlers seems intuitive for simulating retrying, it has a severe limitation: the number of retries attempted is bound by the depth of the nested `try` blocks and moving beyond a depth of two can make program incomprehensible. This limits the programs capability to achieve G3.

To remove the spare handler, apply *Introduce Resourceful Try Clause*. Place code for delivering the requested service, including primary normal actions and alternative actions upon exceptions, inside a selection code construct in the `try` clause. Prepare for retries by applying *Introduce Checkpoint Class* to the catch clause to ensure that the retries are launched in a correct state.

4.3.6.2 Mechanics

- i. Enclose the `try` block in a `do-while` construct.
- ii. Declare variables to control the termination of the `do-while` construct.
- iii. Write a selection mechanism to choose between the primary implementation and the alternative implementation.

- iv. Move code in the spare handler to the associated `try` clause as the alternative implementation.
- v. In the `catch` clause, increase the attempt count by one.
- vi. Throw a failure exception if the retry process reaches the maximum number of attempts specified.
- vii. Compile and test.

4.3.6.3 Example

An access control system reads user's data either from a relational database or from a LDAP server. User's data in the relational database and the LDAP server are automatically synchronized. One of two servers may be shut down for maintenance but the access control system must still function normally. You have a `readUser` method to perform the task which uses a nested `try` block as an alternative:

```
public void readUser(String name) throws ReadUserException {  
  
    try {    readFromDB(name);    /* may throw an IOException */ }  
  
    catch (IOException e) {  
  
        try {    readFromLDAP(name);    /* may throw an IOException */ }  
  
        catch (IOException e) {    throw new ReadUserException(e); }  
  
    }  
  
}
```

Note that the `readUser` method has only two opportunities to fulfill its responsibility: if the access to relational database fails, access to LDAP server is attempted as an alternative. You apply *Introduce Resourceful Try Clause* to remove the

nested `try` block in the `catch` clause and enable the `readUser` method to retry, for example, five times:

```
public void readUser(String name) throws ReadUserException {  
  
    int attempt = 0;        int maxAttempt = 5;    boolean retry = false;  
  
    do {  
  
        try {    retry = false;  
  
            if (attempt == 0)    readFromDB(name);    /* primary */  
  
            else                readFromLDAP(name); /* alternative */ }  
  
        catch (IOException e)    {  
  
            attempt++;        retry = true;  
  
            if (attempt > maxAttempt)  
  
                throw new ReadUserException (e);    }  
  
        } while (attempt<= maxAttempt && retry)  
  
    }
```

Chapter 5 A PATTERN-BASED SOLUTION TO THE JAVA EXCEPTION OVERRIDDEN PROBLEM

5.1 Problem Description

Java's current exception handling mechanism design can easily lead the programmer to throwing exceptions that convey incomplete information regarding the exceptional conditions. In general, an exception is thrown with the coding idiom `throw new SomeException()`, which creates an anonymous instance of the `SomeException` class, makes the anonymous instance active, and unwinds the stack frames in search for a suitable exception handler in the chain of callers. The Java exception handling mechanism is in control of the unwinding process except when a `finally` clause is encountered, in which case the control is transferred. Since new exceptions can be thrown while a `finally` clause is in control and since Java allows at most one active exception at any time, the programmer has the options of propagating either the original active exception or the newly raised exception. Unfortunately, either option causes part of the exceptional information to become lost. Even worse, there is no way for the caller to decipher what really happened. This hampers the caller's capability to accurately deal with exceptions.

Figure 5.1 shows such an example taken from [1]. In the period of time after the declared `BadDataSetException` is thrown (line 08) and before it is propagated (if at all) to the caller, the control is transferred to the `finally` clause for cleanup (lines 09-16). As can be seen, additional exceptions can be encountered in trying to close the input file (line 11). There are two common ways for handling the cleanup exception:

```

01  public double [] getDataSet(String setName) throws BadDataSetException {
02      String file = setName + ".dest";
03      FileInputStream in = null;
04      try {
05          in = new FileInputStream(file);
06          return readDataSet(in);
07      } catch (IOException e) {
08          throw new BadDataSetException();
09      } finally {
10          try {
11              if (in != null) in.close ();
12          } catch (IOException e) {
13              ; // ignore: we either read the data OK
14              // or we're throwing BadDataSetException
15          }
16      }
17  }
18      // ... definition of readDataSet ...

```

Figure 5.1: An example of a lost cleanup exception

ignoring the cleanup exception to prevent the declared `BadDataSetException` from being overwritten, or, alternatively, propagating the cleanup exception by allowing it to overwrite the declared exception. In either way, part of the exception information becomes lost and the execution of `getDataSet` may lead the system to an erroneous state not observable by the caller. This becomes a serious problem, for example, when the caller attempts a retry operation. Retrying an operation while the system is already in an incorrect state will eventually make the situation worse and complicate the debugging process.

Figure 5.2 shows such an example by illustrating an execution path of Figure 5.1. Four actions are concerned:

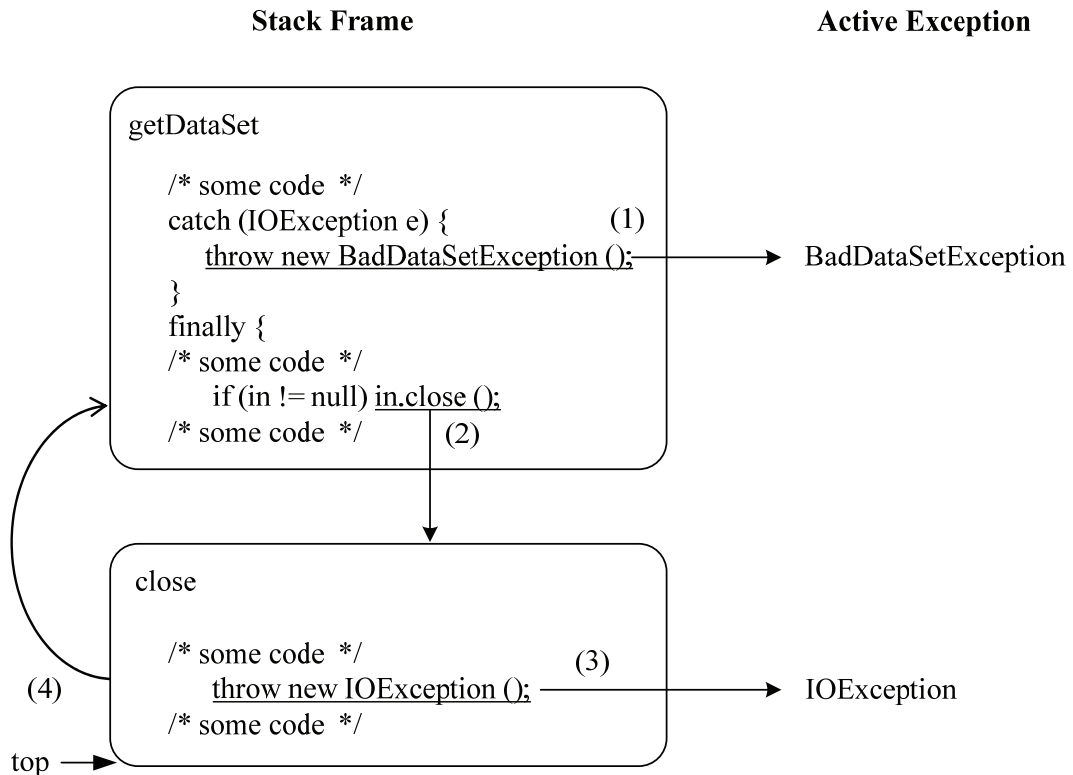


Figure 5.2: An execution path showing the lost of exception problem

1. In the catch clause, the statement `throw new BadDataSetException();` creates and activates an anonymous instance of `BadDataSetException`.
2. The `finally` clause is executed for cleanup. The call `in.close();` pushes a new frame on the top of the stack.
3. The execution of `in.close();` throws an instance of `IOException` which is an active exception on the top stack frame.
4. The `IOException` causes `in.close();` to terminate abruptly; the top stack frame gets popped; and the `IOException` is propagated to `getDataSet`.

How does `getDataSet` deal with an `IOException` encountered in the `finally` clause? Since `IOException` is checked, according to Java's

handle-or-declare rule, there are three plausible options:

- take the “declare” option by declaring `IOException` in the method signature of `getDataSet`. This allows `IOException` to overwrite `BadDataSetException`;
- take the “handle” option by catching `IOException` and re-throwing an unchecked `CleanupException` to indicate cleanup failure. The `CleanupException` overwrites `BadDataSetException`; and
- take the “handle” option by catching `IOException` and ignoring it. This retains `BadDataSetException` as the active exception.

In the first option, the low-level implementation exception `IOException` unjustifiably becomes a declared exception of `getDataSet`. In the second option, the checked `IOException` is wrapped in the unchecked `CleanupException`. It leaves the interface of `getDataSet` unchanged and avoids the interface evolution problem. In addition, `CleanupException` can preserve more information by making `IOException` its root cause with exception chaining.

However, in both the first and the second options, since the cleanup exception is allowed to overwrite the declared `BadDataSetException`, upon receiving a cleanup exception, caller of `getDataSet` is unaware whether the `try` clause was executed successfully. Specifically, it is possible that (1) the `try` clause executed normally and correctly changed the state of the component but the caller thinks otherwise and attempts to perform the method again; or (2) the `try` clause has failed and did not change the state of the component but the caller thinks otherwise. Either way, the caller is led to an erroneous state.

The third option, which is advocated in popular Java texts (e.g., [1]), *represents a tradeoff between accuracy and simplicity*. The declared `BadDataSetException` is preserved by ignoring all cleanup exceptions. However, cleanup exceptions raised by resource disposal methods can indicate a bug in the resource components. Ignoring cleanup exceptions obliterates useful information for fixing such a bug. Further, cleanup exceptions might indicate a resource leak problem. Once such exceptions are ignored, no effective remedy can be administered. Eventually, the resource becomes depleted and the application has to be restarted.

As far as keeping complete exceptional information is concerned, an alternative option is to catch the cleanup `IOException`, wrap it in an unchecked `CleanupException`, attach `CleanupException` to `BadDataSetException`, and retain `BadDataSetException` as the active exception. To do this, we need to be able to access to the `BadDataSetException` in the `finally` clause. This requires a change in the coding style for throwing exceptions; a possible workaround is shown in Figure 5.3. An extra local variable is declared (line 3) which holds a reference to the instance of the declared exception (line 7) so that it can be accessed in the `finally` clause (lines 12-15).

Though the implementation is relatively easy, the workaround shown in Figure 5.3 can end up producing complicated code that is difficult to understand and test. For example, suppose that there are N declared exceptions which are thrown by N `catch` clauses and M resources which are released in the `finally` clause. Thus, there will be N local variables, say DE_1 to DE_N for each of the declared exceptions, respectively. In each of the `catch` clauses, DE_1 to DE_N are set and thrown. Next, in the `finally` clause, there are M `try-catch` blocks to release the M resources; each section of the cleanup code is similar to that on lines 9-18 of Figure 5.3. In addition, a test for finding

```

01  public double [] getDataSet(String setName) throws BadDataSetException {
02      /* some code */
03      BadDataSetException de = null;    // a local variable for the declared exception
04      try {
05          /* some code */
06      } catch (IOException e) {
07          throw (de = new BadDataSetException()); // de becomes active
08      } finally {
09          try {
10              if (in != null) in.close ();
11          } catch (IOException e) {
12              if (de != null) {
13                  CleanupException ce = new CleanupException(e);
14                  /* code to attach ce to the active exception de */
15              }
16              else
17                  throw new CleanupException(e);
18          }
19      }
20  }

```

Figure 5.3: A possible workaround for accessing the active exception

a non-null object (i.e., the active exception) from DE_1 to DE_N must be made so that the cleanup exceptions can be attached to it.

It should be noted that exceptions can occur during other exception handling activities as well. For example, before throwing a declared exception, the programmer may have decided to restore the exception-throwing component to a correct state. As another example, the programmer may have decided to attempt retry. In either example, new exceptions can be thrown. Like the previous example of cleanup exceptions, they need to be collected and attached to the declared exception as well. The exception handling code will become much more complicated if we take all possible types of

subsequent exceptions into account.

5.2 Exception Handling Patterns

In what follows, we propose a pattern-based approach to solving the exception overridden problem. Using patterns for exception handling is not a new idea (e.g., see [32]). However, our patterns cover a problem not previously addressed in exception handling: *how to represent the failure of an operation by preparing an exception that provides complete and unambiguous information on the damage a fault has caused*. In other words, we want to preserve a sequence of exceptional conditions inside a single exception object.

5.2.1 Pattern 1: Exceptions conveying damage information

A component signals an exception to admit its inability to deliver correct services. From the perspective of the caller, useful exception handling design can be conducted only if the types of the exceptions that the caller may encounter are foreseen. However, an exception type alone is insufficient to represent all exceptions that the component has encountered internally. For example, the component may encounter cleanup exceptions which are usually ignored to prevent the declared exception from being overridden. The caller is totally unaware of the ignored cleanup exceptions. Thus, there is nothing the caller can do to cope with this situation.

Therefore, preserve all subsequent exceptions and attach them to the exception which is going to be propagated to the caller. An exception can and should provide initial information on the damage a fault has caused [6]. The caller can then make use of the damage information to refine its exception handling design.

To make an exception convey damage information, three things must be done. First, all subsequent exceptions must be collected. Second, the exception object must have the

ability not only to represent the failure semantics but also to carry all subsequent exceptions to represent the damage information. Third, the collected subsequent exceptions have to be attached to the exception.

Consequences

The consequences of applying this pattern are:

- Declared exceptions are prevented from being overridden.
- Subsequent exceptions are prevented from being ignored.
- Initial damage information that a fault caused can be explored by investigating the exception object, which carries all exceptional conditions.
- Exception handling requires collaboration among the `try`, `catch`, and `finally` clauses, which complicates exception handling programming.

Known use

Collaborative exception handling in distributed computing recommends that all exceptions raised by collaborating processes, threads, or agents be preserved and be organized as a tree [33]. This pattern suggests that subsequent exceptions should be preserved and reported as well even in single-threaded programming.

5.2.2 Pattern 2: Safety Method and Safety Command

We are going to implement the *Exceptions Conveying Damage Information* pattern. First of all, we have to prevent the declared exception from being overridden by any subsequent exception raised by exception handlers or by cleanup operations. For

instance, one way to do this is to enclose the cleanup operations in a nested `try` block, as shown in Figure 5.1, lines 10-15. However, such a nested `try` block yields complicated program structure where exceptional code is interleaved with normal code. The program ends up being difficult to read, maintain, and test [7]. In addition, as the number of resources grows, the number of nested `try` blocks grows as well. As a result, the entire method becomes a *long method* [29].

Therefore, enclose any exception handling activity which may throw subsequent exceptions inside a safety block, which is a try block with a blanket catch (a catch clause catching the Throwable or Exception class). Further, encapsulate the safety block in a method or in a first-class object as Command in the Command pattern [21]. Lastly, replace the original code with invocation to the method or the Command object. The method and the Command that implement this approach are called a Safety Method and a Safety Command, respectively.

```
01 public class CleanerUtility {
02     public static void close(Connection target) {
03         try {
04             if (target != null) { target.close(); }
05         } catch (Exception e) { /* for now, ignoring all exceptions */ }
06     }
07     public static void close(PreparedStatement target) {
08         /* same implementation of the above method */
09     }
10     /* other safety methods */ }
```

Figure 5.4: Overloaded static methods in the `CleanerUtility` class

Implementation

In general, *Safety Methods* are used to dispose of a particular type of resources. *Safety Methods* are usually placed in a static utility class. Thus, the utility can be used without instantiation. The names of these *Safety Methods* are identical and only the parameter types are different. Such static polymorphism technique can simplify the use of the utility while guaranteeing types checking at compile time. Figure 5.4 shows a `CleanerUtility` class which is used to release JDBC resources such as `Connection`, `PreparedStatement`, `ResultSet`, and so on.

It can be seen that the code in each *Safety Method* is identical except the parameter type. One way to prevent the duplication is to use *Safety Command* with *Template Method* pattern [21]. As shown in Figure 5.5 (a), an abstract class `Cleaner` is designed to encapsulate the implementation of a safety block in the template method `close` (lines 4-8), which delegates the actual cleanup operation to the primitive operation `closeResource`. A *Safety Method* now is promoted to a subclass of `Cleaner`. For example, Figure 5.5(b) shows a `ConnectionCleaner` class which is used to release JDBC `Connection` object. The implementation of `ConnectionCleaner` is straightforward. It only defines a constructor to accept the resource object and overwrites the primitive operation to dispose of the resource. The results of applying patterns in these two designs are shown in Figure 5.6.

```

01 abstract public class Cleaner <T>{
02     private T target = null;
03     public Cleaner(T target) { this.target = target; };
04     public void close() {    /* template method */
05         try {
06             if (target!= null) {    closeResource(); }
07             catch (Exception e) { /* for now, ignoring all exceptions */    }
08         }
09     protected abstract void closeResource() throws Exception;
                                   /* primitive method */
10     protected T getTarget() { return target; }
11 }

```

(a)

```

01 public class ConnectionCleaner extends Cleaner<Connection>{
02     public ConnectionCleaner(Connection target) { super(target); };
03     protected void closeResource() throws SQLException{
04         getTarget().close();
05     }
06 }

```

(b)

Figure 5.5: The source code of (a) Cleaner and (b) ConnectionCleaner.

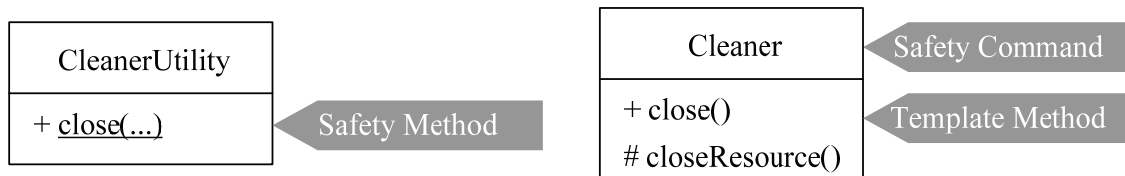


Figure 5.6: Patterns applied in `CleanerUtility` and `Cleaner` classes

Consequences

The consequences of applying *Safety Method* and *Safety Command* are:

- Both checked and unchecked cleanup exceptions (except those of type `Error`) are caught as a means to prevent the declared exception from being overridden.
- Nested `try` blocks are avoided and code in the `finally` clause is easy to understand and test.
- Cleanup operations for each resource type can be reused.
- *Safety Method* is easier to use than *Safety Command* because no object instantiation is required.
- Cleanup exceptions are inevitably ignored which yields the problem of ambiguous semantics of declared exceptions. Although cleanup exceptions can be logged, understanding the problem caused by the exception from the log file usually requires manually inspection at maintenance which is time-consuming and labor-intensive.
- *Safety Method* violates the *open-closed principle* [23] since the static utility class has to be modified once a new type of resources is added to the application. An alternative is to implement a *Reflective Safety Method* which

applies Java Reflection API to invoke the resource disposal method on an object. One *Reflective Safety Method* can be used to dispose of all types of resources at the cost of execution time overhead.

- *Safety Command* satisfies the open-closed principle but might yield many small classes which are very similar.

Known use

Regardless of the problem of ignoring cleanup exceptions, enclosing cleanup code in *Safe Methods* is a common exception handling practice. For example, JDBC Recipes uses a `DatabaseUtil` to close different types of JDBC resources such as `Connection`, `Statement`, and so on [34]. Java Puzzlers demonstrates such an approach by utilizing the new `Closeable` interface of Java release 5.0 to close I/O streams [35].

5.2.3 Pattern 3: Exception Collector

A problem with the *Safety Method* and *Safety Command* is that captured subsequent exceptions are ignored. To fix this problem, The *Collecting Parameter* pattern [36] is used to collect exceptions raised inside each *Safety Method* or *Safety Command*. *Collecting Parameter* solves the problem: “How do you return a collection that is the collaborative result of several methods?” *Collecting Parameter* recommends adding a parameter to each of the methods and passing an object to collect the results.

```

01  public interface IExceptionCollection {
02      void add(Exception e);
03      void clear();
04      int size();
05      Exception[] toArray();
06  }

```

Figure 5.8: The source code of IExceptionCollection

```

01  public static void close(Connection conn, IExceptionCollection collection) {
02      try {
03          if (conn!= null) {
04              conn.close();  }
05          } catch (Exception e) {
06              collection.add(e); /* collect an exception */
07          }
08      }
09  }

```

Figure 5.7: A Safe Method applying Collecting Parameter

Therefore, create an Exception Collector class implementing the Collecting Parameter to collect subsequent exceptions. In implementation, we create a new interface IExceptionCollection to collect the exceptions raised during the execution of *Safety Method* and *Safety Command*. Figure 5.8 shows the code for the IExceptionCollection.

With *Exception Collector*, *Safety Method* and *Safety Command* will no longer ignore cleanup exceptions. Figure 5.7 is a revision of *Safety Method* implementing *Collecting Parameter* where the new code is underlined. The results of applying *Collecting Parameter* in *Safety Method* and *Safety Command* are shown in Figure 5.9.

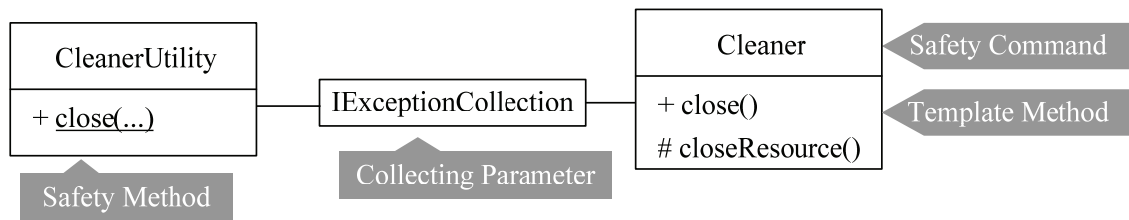


Figure 5.9: *Safety Method* and *Safety Command* applying *Collecting Parameter*

Known use

Collecting Parameter has been used in many applications. JUnit framework uses a *Collecting Parameter* `TestResult` to gather results from a number of test cases [37]. The Eclipse platform applies a *Collecting Parameter* `IProgressMonitor` to inform users about progress of a set of operations [37].

5.2.4 Pattern 4: Exception with Variable State

With *Exception Collector*, *Safety Method* and *Safety Command* can collect subsequent exceptions. Suppose that we have held a reference to the active exception. The question is how to attach the subsequent exceptions to the active exception.

The design of the Java exception classes does not encourage the attachment of subsequent exceptions. The only exception that can be attached to the Java built-in exception object is the cause of the exception. The cause of an exception is used to chain a sequence of exceptions to preserve stack traces. Keeping the stack trace is important because it is useful for error diagnosis and debugging. To convey subsequent exceptions, Java exception classes must be extended.

Extending Java exception classes is generally achieved by subclassing (e. g., see [28]). You can inherit a Java built-in exception class, add extra fields to it, and implement corresponding setter and getter methods to store and retrieve the desired data.

```

01  public interface IVariableState <E, V>{
03      V getData(E key);
04      void setData(E key, V value);
05      Set<E> keySet();
06      Collection<V> values()
07  }

```

Figure 5.10 IVariableState defining the interface of *Variable State*

However, subclassing a class merely to add some fields is overkill, tedious, and violates general object-oriented principles. For example, even if the Java `IOException` is satisfactory for your applications, to carry out application-defined information you still have to subclass `IOException`. Additionally, doing so tends to lead to lots of new exception classes whose semantics are similar to the Java built-in exception classes, which is not recommended [38].

The *Variable State* pattern, which is an alternative for subclassing, fits our needs. It solves the problem “How do you represent state whose presence varies from instance to instance? [36]”

Therefore, implement Variable State in your exception classes by create an IVariableState interface defining necessary methods of Variable State; see Figure 5.10. Design a checked ApplicationException and an unchecked EExceptionHandler (i.e., exception handling exception) to represent the root of the exception classes in your programs; implement the IVariableState in the root class to avoid unnecessary subclassing.

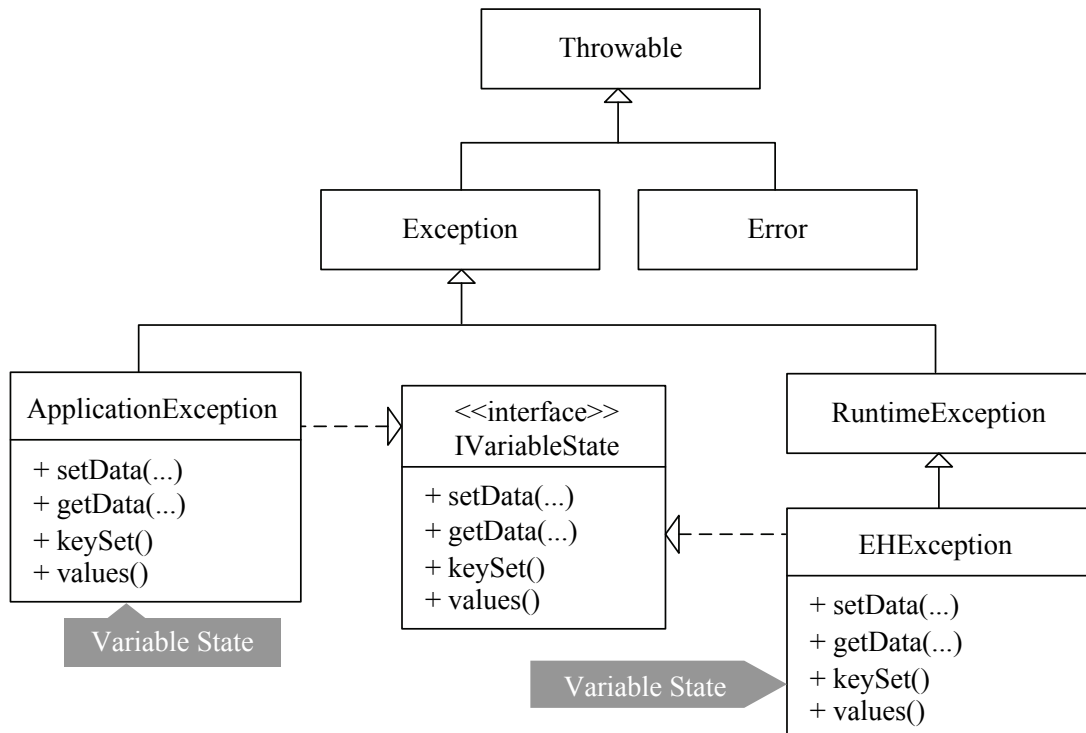


Figure 5.11 The exception hierarchy implementing *Variable State* pattern

To implement *Variable State* for the `ApplicationException` and `EHEException` classes, add a data member that implements the `Map` interface and provide setter and getter methods for the map. You can set and access subsequent exceptions and any other information to the exception object under a particular key. By so doing, all you have to do to attach extra information to existing exception objects is to define the protocol (the key of `String` type) to set and get the data. Note that to facilitate accessing all the data in the map, we allow returning all keys and objects stored in the map via `keySet` and `values`, respectively. The results of applying *Variable State* in the `ApplicationException` and `EHEException` classes are shown in Figure 5.11.

Consequences

The consequences of applying *Variable State* on exception class design are:

- Unnecessary subclassing of exception classes is avoided.
- Fostering the reuse of existing exception classes. For example, it is not necessary to devise a `MyIOException` as a replacement of Java built-in `IOException` merely to carry extra data.
- Exception overriding and ignoring problems are avoided since subsequent exceptions can be attached to the active exception.
- An exception object can convey as much as possible information to represent the damage caused by a fault since an exception created at runtime forms an exception tree rather than an exception chain. The temporal order of the attached exceptions can be determined by applying particular naming convention on the key (e.g., adding a sequence number at the tail of the key) or by adding a timestamp in the attached exceptions.
- Setting and getting data via a string-based key requires extra care by programmers. For example, accessing a data which is not set yet will cause a runtime exception.
- Unless Java `Throwable` class implements *Variable State*, there is no straightforward method to attach data to Java built-in classes.
- The meaning of exception tree traversal is not clear. For example, discovering an instance of `IOException` indicates nothing but a subsequent exception. You cannot tell whether it represents a failure of cleanup or a failure of error recovery operations.

Known use

Variable State is implemented in Visual Smalltalk, Standard Widget Toolkit (SWT), and JFace [37]. The `Exception` class in the Microsoft .NET framework supports *Variable State* with a `Data` attribute [39].

5.2.5 Pattern 5: Exception Hierarchy for Abnormal Behavior

By implementing *Variable State*, our exception objects can convey any number of subsequent exceptions of any type. However, most existing exceptions are designed and structured to represent failures of normal operations. In the context of subsequent exceptions, what we need is an exception hierarchy that can represent failures of cleanup and exception handling activities.

For example, exceptions such as `IOException` and `SQLException` can be raised by the normal operations in the `try` clause, the error recovery operations in the `catch` clause, and the cleanup operations in the `finally` clause. If such “raw” exceptions are directly attached to the active exception, how can the receiver differentiate an error recovery exception from a cleanup exception if both of them are instances of `IOException`? The receiver inevitably has a difficulty in interpreting the meanings of these raw exceptions.

Therefore, design an unchecked exception hierarchy to represent the failure of each category of exception handling activities. For each category of exception handling activities, create a corresponding exception class to indicate its failure. In implementation, prepare an instance of an exception class from the exception hierarchy by attaching raw exceptions to it and then attach the prepared exception to the active exception.

Figure 5.12 illustrates an exception hierarchy representing failures of common exception handling activities [6][40]:

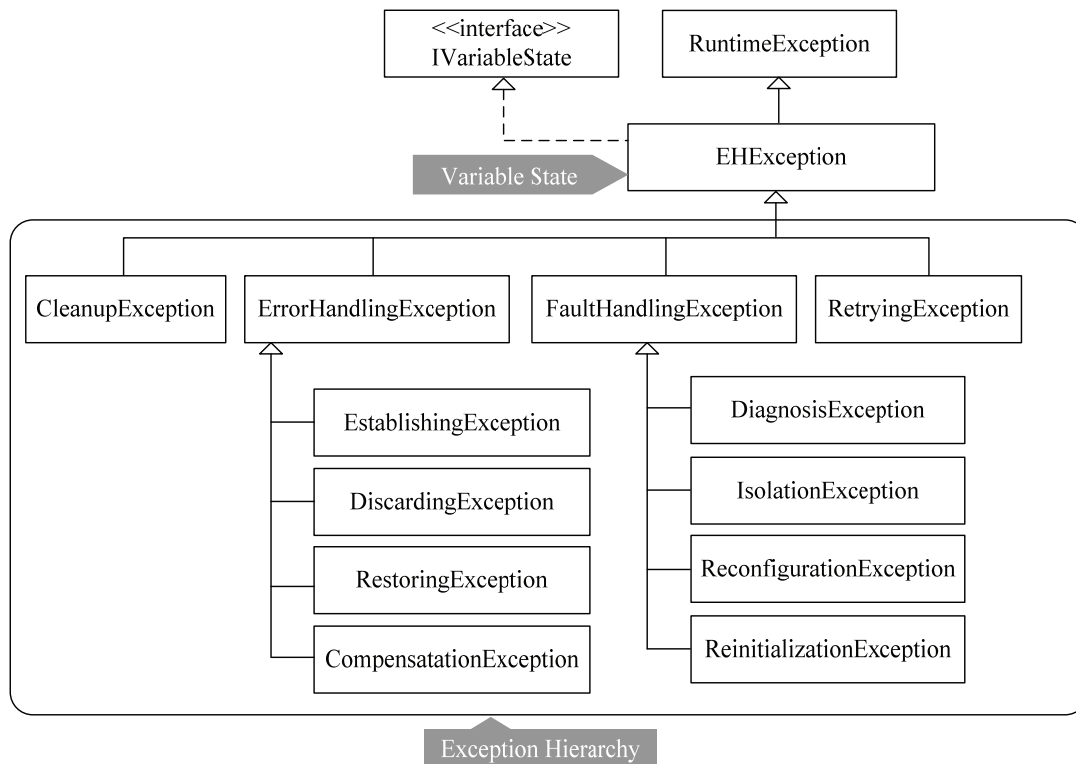


Figure 5.12: An exception hierarchy for exceptions of exception handling activities

- `CleanupException` indicates failures in disposing resources in the `finally` clause.
- `ErrorHandlerException` represents failures of restoring a component from an erroneous state to a normal state. When backward error recovery is utilized, the first three subclasses represent the refined failures in initiating a recovery region, in dropping the recovery region, and in recovering to a prior state with the recovery region, respectively. `CompensationException` represents failures of undo operations.
- `FaultHandlingException` represents failures in steering a component away from the cause of errors so that the intended service of the component is delivered. Its subclasses represent each step involved in a fault handling

process.

- `RetryingException` represents failures in previous attempts before making the current retry.

Consequences

The consequences of classifying subsequent exceptions with a well-defined exception hierarchy are:

- The semantics of an exception tree created at runtime is unambiguously defined with respect to the failure on each exception handling activity. In other words, the exception tree provides the real exceptional postconditions at runtime when a method returns exceptionally.
- The exception tree facilitates debugging because all exceptions are kept in the tree.
- The exception tree facilitates testing. If the exception tree frequently conveys error handling, fault handling, and/or cleanup exceptions, it might be a sign of poor quality of the production code. More test cases can be derived from the information conveyed by the exception tree.
- The exception tree enables us to devise sophisticated exception handling routines to enhance system robustness. For instance, if a failure exception includes error handling and/or cleanup exceptions, it is clear that the component is in an erroneous state. Thus, the caller has to make extra efforts to restore to a correct state.
- End users can benefit from the information stored in the exception tree. For

example, end users are not allowed to attempt a retry operation if error handling exceptions are attached to the exception tree, which implies that the system has been in an erroneous state. Retrying an operation while the system is already in an incorrect state will eventually make the situation worse and complicates debugging.

- To construct the exception tree, exception handling activities must be coordinated, which requires extra efforts.

Known use

Exception Hierarchy is a common exception handling pattern [28][41]. Programming languages such as Java and C# have built-in exception class hierarchies. Developers are encouraged to construct application-dependent exception hierarchies to represent application level exceptions.

5.2.6 Pattern 6: Thrower

With exception classes now implementing *Variable State*, subsequent exceptions can be attached. However, since Java does not allow accessing the active exception in the `finally` clause, we must find a way to attach these subsequent exceptions to it.

One possible way is to set a local variable to represent the active exception before throwing it. Then, collect subsequent exceptions and attach them to the active exception via the local variable. This leads to the setting, throwing, collecting, and attaching (STCA) process of exception throwing regarding subsequent exceptions. Unfortunately, this process does not work because throwing an exception in the `catch` clause transfers the control immediately to the `finally` clause. As a result, error handling code placed after the `throw` keyword is unreachable. Hence, it is impossible to execute error


```

01 public void readUser(String name) throws ReadUserException {
02     int attempt = 0;
03     ReadUserException ex = null;
04     do {
05         try {
06             ex = null;
07             if (attempt < 2)
08                 readFromDB(name);    /* may throw an IOException */
09             else
10                 readFromLDAP(name); /* may throw an IOException */
11         } catch (IOException e) {
12             attempt++;
13             ex = new ReadUserException (); /* set an exception */
14             /* code for error handling and collecting follow exceptions */
15             /* code for attaching follow exceptions to ex */
16             throw ex;
17         } finally {
18             /* code for cleanup and collecting cleanup exceptions. */
19             /* code for attaching cleanup exceptions. */
20         }
21     } while(attempt <= 5 && (ex != null));
22 }

```

Figure 5.13: The SCATCA process for retry

handling code and to collect error handling exceptions together.

To collect subsequent exceptions raised by error handling activities, we may revise the STCA process to one of setting, collecting (for error handling exception), attaching, throwing, collecting (for cleanup exceptions), and attaching (SCATCA) process.

However, it only works for a `try` block which is executed one time only. SCATCA does not work when the common exception handling option `retry` is applied. Consider the code fragment shown in Figure 5.13. The `readUser` method tries up to five times to fulfill its responsibility. In the first two times, it reads user data from a relational database (line 8). In the others, a LDAP server is used instead (line 10). Each of the two operations can throw an `IOException`.

A `do-while` loop is used to implement `retry`. The exception throwing process follows SCATCA process. However, when the `ex` exception is thrown on line 16, the method returns exceptionally after executing the `finally` clause. The enclosed `try` block (lines 5-19) does not get re-executed. That is, the `do-while` loop has no effect.

Due to the fact that throwing an exception immediately changes the control flow which tends to complicate exception handling implementation, whether a code block will cause an exception and what information it will contain should only be determined after the code block is completely executed.

*Therefore, replace the `throw` statement with a process of setting, zero or more repetitions of collecting (for any subsequent exception) and attaching, and conditionally signaling ($S[CA]*C-S$). Design a `Thrower` class to encapsulate these operations (see Figure 5.14). The behavior of `signalIfError` method depends on the active, cleanup, and other subsequent exceptions. In a nutshell, if there is no exception, `signalIfError` does nothing; if there is an active exception, it will be propagated with any subsequent exceptions (including cleanup exceptions) attached; and if there is only a cleanup exception, it will be propagated directly to indicate a cleanup bug.*

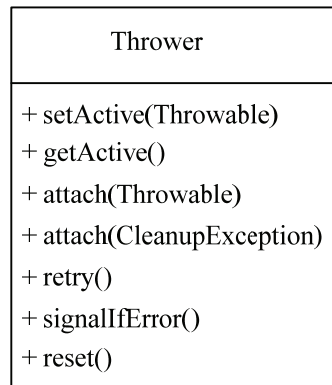


Figure 5.14: The *Thrower* class

Consequences

The consequences of applying *Thrower* to signal exceptions are:

- Making the sequence of exception handling activities be logical. Ideally, the `throw` keyword instantly leaves a `try` block and the code after it is unreachable. However, since the `finally` clause is always executed, the program state before and after a `throw` keyword might be changed and thus odd program behavior can be caused [5]. For example, the return value might be overridden in the `finally` clause [1]. By replacing the `throw` keyword with a conditionally signaling instruction at the end of a code block, the execution sequence aligns with the code structure.
- Encapsulating exception production. The process to signal an exception with subsequent exceptions is encapsulated.
- Supporting retry. As shown in Figure 5.15, the faulty exception handling code for retry in Figure 5.13 can be fixed.
- Complicated state-dependent behavior. Conditionally signaling an exception

causes state-dependent behavior which might not be easy to implement and understand.

- Degrading compile-time checking for checked exceptions which are thrown but not declared. The `signalIfError` method does not declare any checked exception, although it can actually throw one. On the one hand, `signalIfError` cannot declare any specific checked exception because it is used by a variety of methods, each of which might throw different types of checked exceptions. On the other hand, however, the Java compiler does not allow a method to throw checked exceptions without declaration. To solve this problem, we use the Java `Unsafe` utility class to raise checked exceptions without declaring them [42]. By so doing, however, methods depend on developers to correctly specify the checked exceptions thrown by `signalIfError`.

Known use

Thrower implements the novel exception throwing idiom suggested in this dissertation. The new exception throwing process involves four steps: setting, collecting, attaching, and conditionally signaling where collecting and attaching may be repeated.

```

01  public void readUser(String name) throws ReadUserException {
02      int attempt = 0;
03      Thrower thrower = new Thrower();
04      do {
05          try {
06              if (attempt) > 0
07                  thrower.retry();
08              if (attempt < 2)
09                  readFromDB(name);           // might throw an IOException
10              else
11                  readFromLDAP(name);        // might throw an IOException
12          } catch (IOException e) {
13              attempt++;
14              thrower.setActive(new ReadUserException ());  /* set an exception */
15              /* code for performing error handling and collecting subsequent exceptions */
16              thrower.attach(/*subsequent exceptions */) /* attach subsequent exceptions */
17          } finally {
18              /* code for cleanup and collecting cleanup exceptions. */
19              thrower.attach( /* cleanup exception */) /* attach cleanup exceptions */
20          }
21      } while(attempt <= 5 && (thrower.getActive() != null));
22      thrower.signalIfError();
23 }

```

Figure 5.15: The revision of exception handling with retry

5.2.7 Pattern 7: State

The behavior of `Thrower` is state-dependent. For example, `signalIfError` acts differently based on whether an active exception is set and subsequent exceptions are attached. In addition, it is illegal to set an active exception with `setActive` after an instance of `CleanupException` has been attached with `attach`. Setting an active exception while one already exists is not valid either. Implementation of such state-dependent behavior is usually complicated and error-prone.

The *State* pattern which “allows an object to alter its behavior when its internal state changes” provides a possible solution [21].

Therefore, identify the state-dependent behavior of `Thrower`; design a state interface to represent the state-dependent behavior; and write a concrete class to represent each state.

The constructed state diagram is represented in Figure 5.16, which includes eight states and ten valid transitions. Figure 5.17 shows the corresponding class diagram implementing this state model. As can be seen in Figure 5.16, initially `Thrower` is in the Normal state. If an active exception is set by invoking the `setActive` method, `Thrower` moves to the Activated state. At this state, any call to the `attach` method leads `Thrower` to the Followed state, indicating that at least one subsequent exception has been attached. `Thrower` remains in the Followed state regardless of how many times the `attach` method is called. In the Normal state, if the `attach` method is invoked, `Thrower` moves to the CleanedUp state.

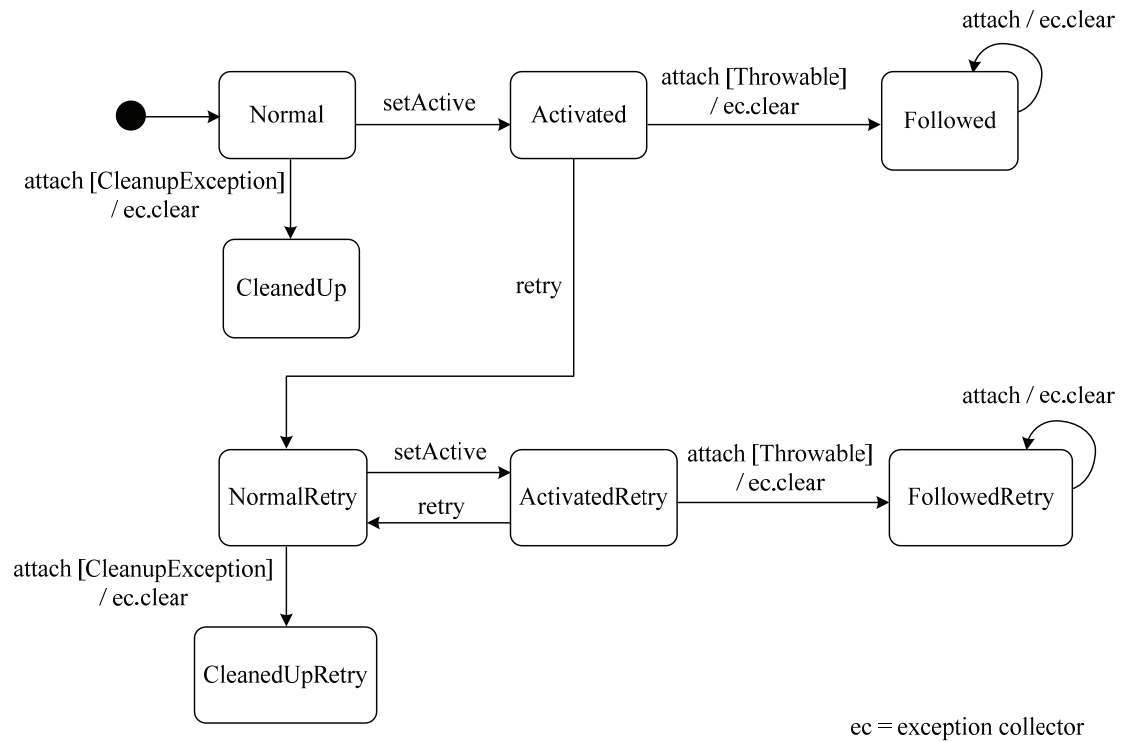


Figure 5.16: A state diagram for the `Thrower` class

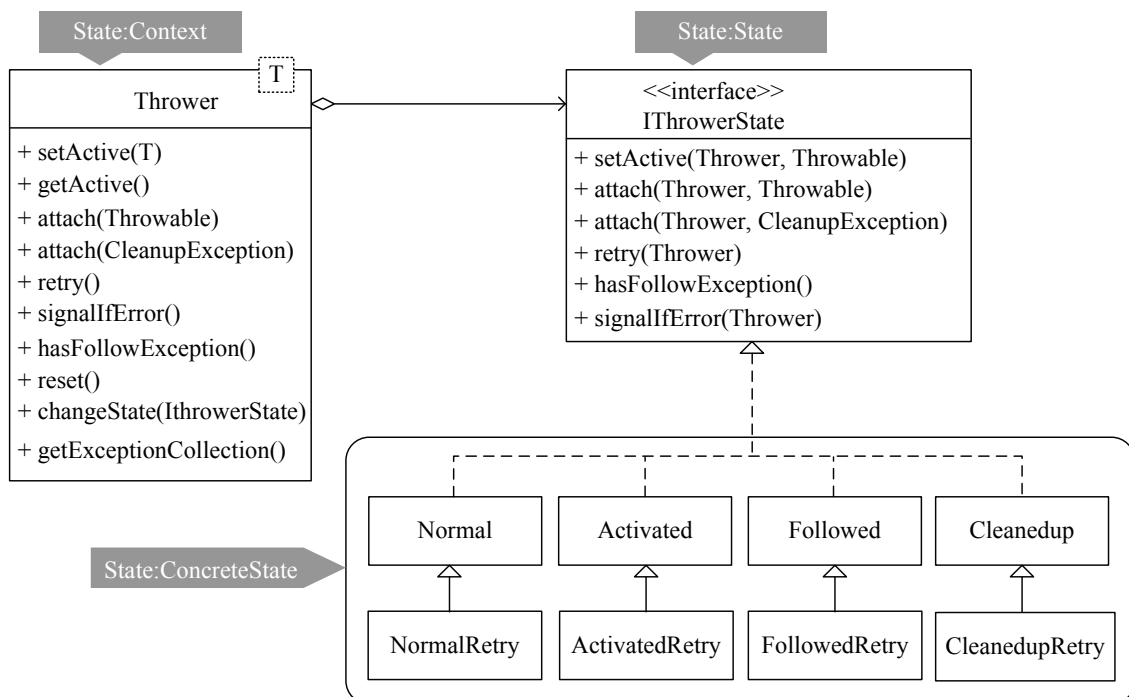


Fig. 17.

Figure 5.17: A class diagram of the `Thrower` and its states

Table 5.1: The behavior of `signalIfError` for each state

State	The behavior of <code>signalIfError</code>
Normal	Doing nothing
Activated	Throwing a declared exception
Followed	Throwing a declared exception with subsequent exceptions
CleanedUp	Throwing a <code>CleanupException</code>
NormalRetry	Doing nothing but might log retry exceptions
ActivatedRetry	Throwing a declared exception with retry exceptions
FollowedRetry	Throwing a declared exception with subsequent and retry exceptions
CleanedUpRetry	Throwing a <code>CleanupException</code> with retry exceptions

It should be noted that the `attach` method is overloaded. In the Normal state, invoking the `attach` method with `CleanupException` triggers a state transition; if the parameter is not `CleanupException`, it is an illegal state transition and an `IllegalStateException` is raised. The state transitions of the bottom half of the state diagram govern behavior during retry and are similar to the top half. The behavior of `signalIfError` in each state is described in Table 5.1.

To simplify the declaration and instantiation of `ICollection<Exception>`, `Thrower` has an instance variable `ec` of type of `ICollection<Exception>`, which can be passed to *Safety Method* and *Safety Command* to collect subsequent exceptions.

When the `attach` method is invoked, `ec` is cleared after the collected subsequent exceptions have been processed. Thus, although there are many types of subsequent exceptions to be collected, one instance of `ICollection` is sufficient.

Consequences

The consequences of applying *State* pattern in `Thrower` are:

- State-specific behavior is extracted from `Thrower` and placed in different concrete state subclasses. Localizing state-specific behavior avoids conditional statements to scatter throughout `Thrower`.
- Legal and illegal state transitions become explicit which simplifies the implementation of concrete state classes.

5.2.8 Pattern 8: Conversion

As shown in Figure 5.18, an exception such as `IOException` gathered by `ICollection` is the raw form of subsequent exceptions. To be attached to the active exception, such a raw exception must be transformed into one of the predefined exceptions shown in Figure 5.12 according to the performed exception handling activity.

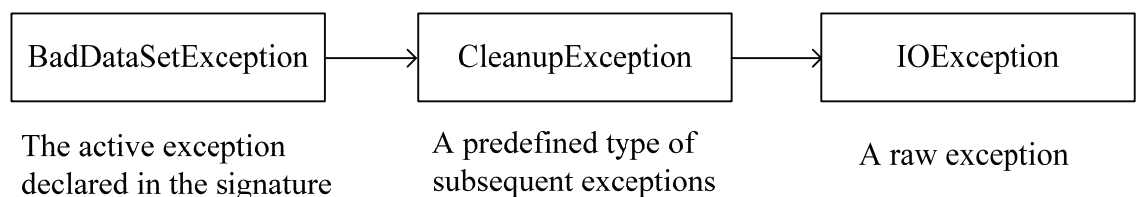


Figure 5.18: Relationships between active, subsequent, and raw exceptions

The *Conversion* pattern which “converts from one object to another [36]” fits this need. There are two types of *Conversion*: *Converter Method* and *Converter Constructor Method* [36]. The former requires the object to be converted (i.e., the class implementing `ICollection`) to provide a method which converts itself into the target object; the latter adds a constructor to the target class accepting the object to be converted as a parameter (i.e., `ExceptionHandler` class and its descendants).

Therefore, add a *Converter Method* `toException` (see Figure 5.19a) to `ICollection` and add a *Converter Constructor Method*, which accepts an instance of `ICollection` as an argument (see Figure 5.19b), to `ExceptionHandler` and all its subclasses. Figure 5.20a and Figure 5.20b respectively show the corresponding class diagram of `ICollection` and `ExceptionHandler` and its subclasses after applying these patterns.

Known use

Conversion is widely used in the Java development toolkit (JDK). For example, the `String` class in Java has six *Conversion Methods* including `toCharArray`, `toLowerCase`, and so on. It also has *Conversion Method Constructors* such as `String(byte[] bytes)` and `String(StringBuffer buffer)` which create a new `String` object from a byte array and a `StringBuffer` object, respectively.

```

01 public EHException toException (Class<? extends EHException > cls cls) {
02
03     EHException follow = null;
04     try {
05         follow = (EHException) cls.getClass().newInstance();
06     } catch (InstantiationException e) { throw new RuntimeException(e)}
07     } catch (IllegalAccessException e) { throw new RuntimeException (e)}
08         for (int i = 0; i < size(); i++)
09             follow.setData(IExceptionProtocol.ELEMENTS, get(i));
10         return follow;
11 }

```

(a)

```

01 public EHException (IExceptionCollection coll) {
02     for (int i = 0; i < coll.size(); i++)
03         setData(IExceptionProtocol.ELEMENTS, coll.get(i));
04 }

```

(b)

Figure 5.19: Source code of (a) IExceptionCollection and (b)

EHException

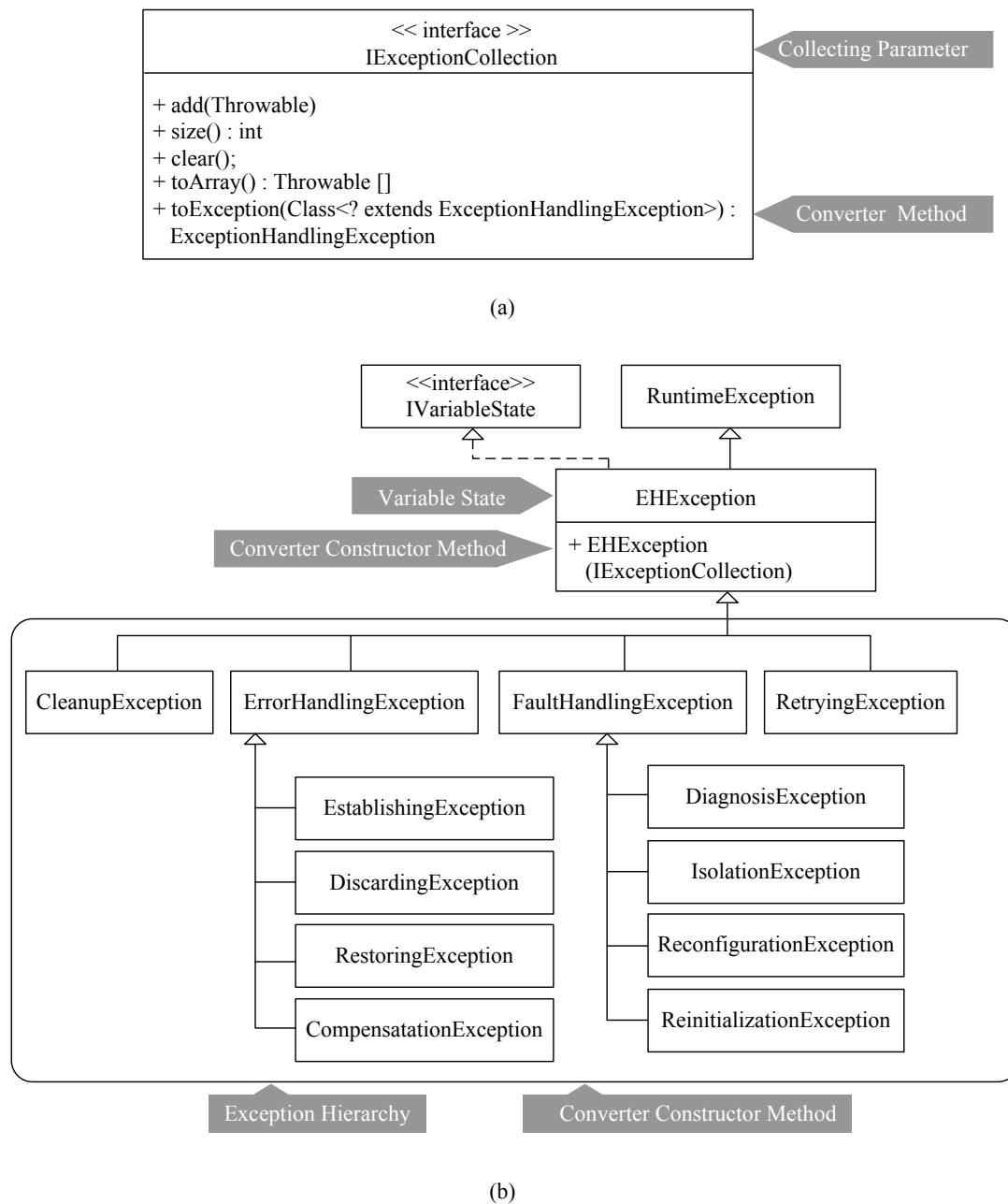


Figure 5.20: (a) `IExceptionCollection` implementing *Converter Method* pattern, (b) `EException` and all its subclasses implementing *Converter Constructor Method* pattern

5.3 Summary of the patterns

The process of applying the proposed patterns is captured by a *pattern language* in which patterns are represented as a directed acyclic graph. In this graph, a pattern defines a context of the patterns that come under it. In this context, a number of forces exist; immediately below the given pattern are the patterns that constitute its principal components, which collectively resolve the forces. Such a process is first introduced by Christopher Alexander and his colleagues [43][44][45] and is adapted in software to generate architectures [37][46].

As shown in Figure 5.21, the presented patterns can be characterized into two categories: high level patterns for exception handling design and low level patterns for implementing the design. The high level patterns are illustrated as a pattern language with annotations of low level patterns. Together, the patterns use an exception object to represent the firsthand damage information regarding the failure of an operation. The sequence of applying these patterns, one at a time, guides the implementation process.

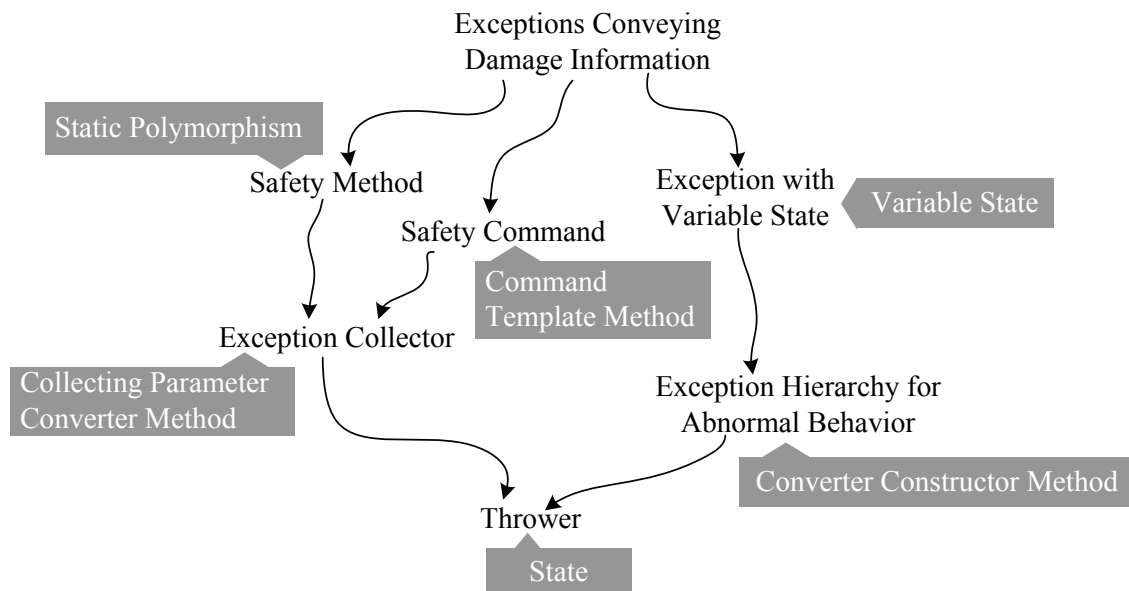


Figure 5.21: A pattern language representation of the proposed patterns

Chapter 6 APPLICATIONS

Based on the robustness model and the exception handling refactorings, we have conducted two case studies and developed two tools as a means to evaluate the usefulness and effectiveness of the research. Chapter 6.1 and Chapter 6.2 show the case studies in detail. Chapter 6.3 and Chapter 6.4 briefly introduce the tools; The development details of the tools were described elsewhere [47][48][49].

6.1 The Robustness Model in Practice

To illustrate the use of the robustness model and the implementation cost for exception handling, we present three hypothetical development scenarios adapted from a real software project.

Suppose that you are in charge of developing a new continuous integration system for Java applications called JCIS. The application will be on exhibition at a trade show in three months. Competition from other tool vendors is fierce and you cannot afford to miss the show. On the other hand, you have to take into consideration that users anticipate good software robustness in the long run. Therefore, you want your application to have good exception handling capability eventually.

Your team practices iterative software development. Your strategy is simple: concentrate on building the core functionalities in the initial iterations, which are code-named the time-to-market (TTM) iterations, to meet the deadline. To do this, the team will not be actively handling the exceptions other than to report and log them. After the initial release, your team will start to make the application more dependable by handling exceptions that are uncovered during the TTM iterations or reported by users. The latter iterations are code-named the robustness iterations. Assume that you successfully made the trade show and received good initial reactions from a number of

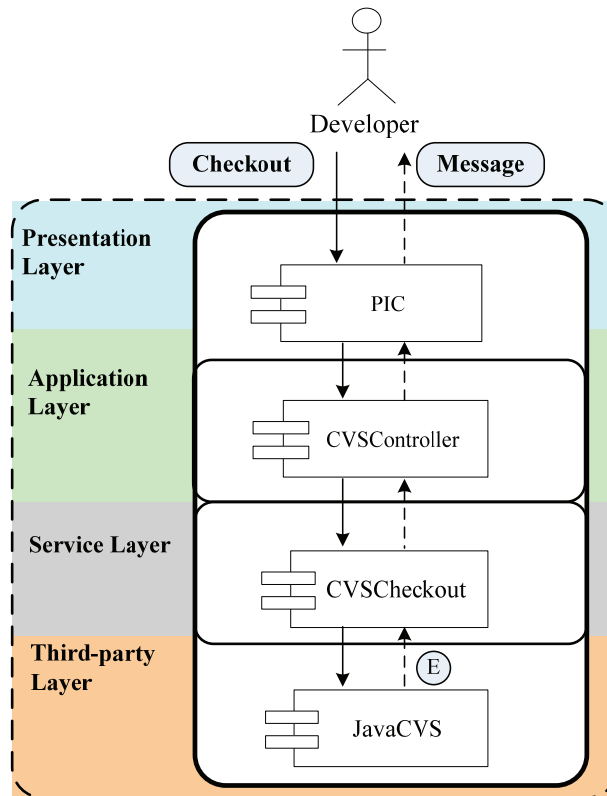
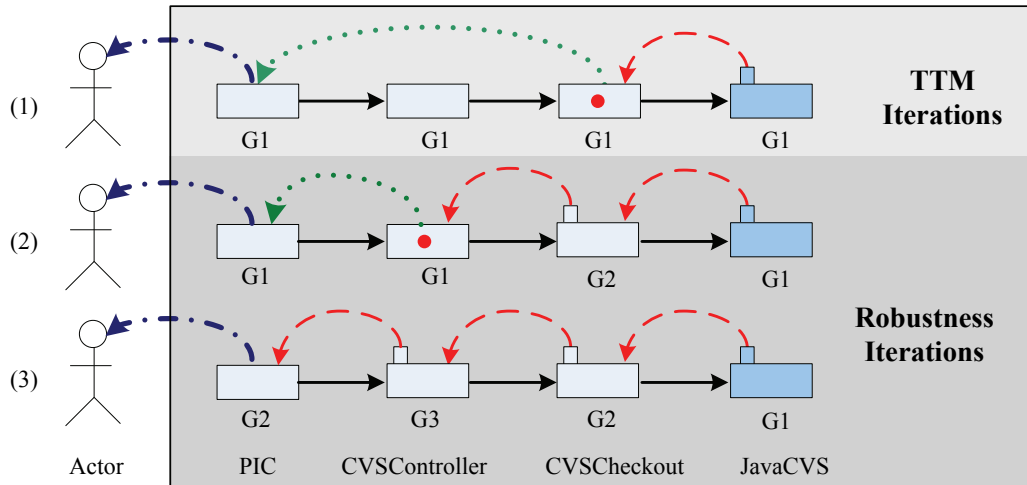


Figure 6.1: Layered architecture of JCIS

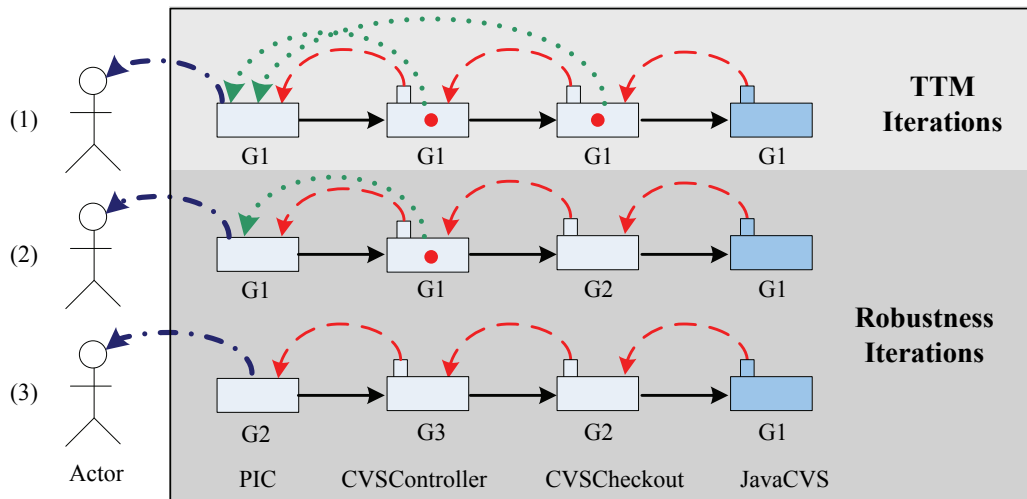
software shops. The project got funding to go on and it's time to do better.

To keep the discussion concise, we consider the occurrence of a networking fault in the real use case of checking out files from a CVS repository for integration. Note that the continuous integration system is a layered system; see Figure 6.2.

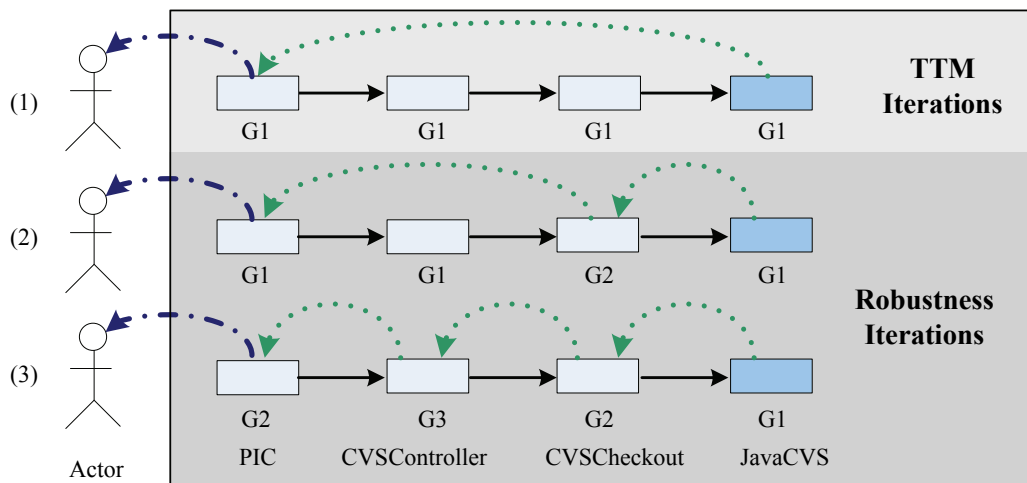
Developer issues a checkout command, which is received by Project Information Center (PIC), a presentation layer component. PIC then delegates the command to CVSController, a controller that manages a variety of CVS commands. In response, CVSController creates a CVSCheckout command object and oversees it to perform the checkout operation. The implementation of CVSCheckout is realized by JavaCVS, an open source third-party library from NetBean, which actually communicates with the CVS repository and handles low level CVS commands. Now, considering the exceptional scenario:



(a)



(b)



(c)

Figure 6.2: Three scenarios of exception handling evolution

Network connection disruption: JavaCVS has established connection to and is currently downloading files from a CVS repository. Network connection disruption causes JavaCVS to throw a checked `ResponseException`.

Figure 6.2 shows three different evolutionary scenarios to cope with the exception. The first two scenarios are applicable where both checked and unchecked exceptions are used (e.g., Java); the third scenario is applicable in situations where only unchecked exceptions are used (e.g., C#). The solid arrow, dashed arrow, and dotted arrow indicate a method call, a checked and an unchecked exceptional return, respectively. Note that logically, an uncaught unchecked exception is directly propagated to the system's top-level component. The dash-dotted arrow denotes a message conveyed to an external actor (e.g., a user or another system). The label under each component is its robustness goals. A small rectangle on the upper-left side of a component represents a declared checked exception on the interface. Finally, the solid circle at the center of a component represents an exception hole, which means that a checked exception is not yet properly handled but directly wrapped to the unchecked `UnhandledException`.

6.1.1 The first scenario

In the TTM iterations, you focus on the application's normal behavior and reactively deal with exceptions. Thus, your application does not declare any checked exception except for the third-party component JavaCVS. By so doing, you bypass the Java's handle-or-declare rule; the developer can then concentrate on the normal functionalities. Figure 6.2 (a.1) is the initial condition where `CVSCheckout` catches `ResponseException` thrown by JavaCVS and wraps it to an `UnhandledException`, which is directly propagated to PIC (cost of wrapping and signaling). The code for exception wrapping marks an exception hole in `CVSCheckout` to be fixed in the robustness iterations.

In the robustness iterations, your effort is directed towards exception handling. You do this by fixing the exception hole in `CVSCheckout`. Since `CVSCheckout` is a command object that does not have enough context information to attempt retry, you make it G2 so that it can be called again in a later retry attempt. To do this, the checked `CVSCheckoutException` is declared in the interface (cost of interface evolution). The following tasks are performed when a `ResponseException` is caught by `CVSCheckout`: (1) the files in the local workspace prior to checkout must be restored (cost of error-handling). To accomplish this, a backup (i.e., a checkpoint) of the local workspace is made before calling `JavaCVS`. If no exception is raised, the backup is dropped. (2) The resource `JavaCVS` is released (cost of cleanup). (3) The caught `ResponseException` is wrapped and thrown as the newly declared `CVSCheckoutException` (cost of wrapping and signaling). As shown in Figure 6.2 (a.2), this pushes the exception hole up to `CVSController`.

`CVSController` now has the exception hole. Since it is an application layer component that has enough context information to retry the failed operation, you upgrade it from G1 to G3 to attempt retry. The following tasks are performed: (1) Declare the checked exception `RepositoryException` in the interface. (2) Design the selection mechanism for enacting a suitable alternative (cost of selection mechanism). (3) Design and implement the alternatives to replace the failed operation (cost of alternative). Note that `CVSCheckout` can participate in the retry since it is G2. (4) In case the retry fails, the caught exception is re-thrown as a `RepositoryException` (cost of wrapping and signaling).

Note that with `CVSController` now at G3, PIC becomes G2 since its state dependency is restricted to `CVSController`, which is now in a correct state even if the retry failed. This means that upon seeing a failure dialog, the user can safely retry the

failed checkout if necessary.

6.1.2 The second scenario

In the first scenario, the exception hole ripples through the architecture layers in a bottom-up manner, which may be regarded as a bad practice because it causes a sequence of interface changes along the reverse call chain. To avoid the ripple effect, you can proactively practice designing for recovery [19], e.g., by exploring the failure scenarios and failure points in each user case [10]. In so doing, `CVSCheckoutException` and `RepositoryException` are identified and specified in the interfaces of `CVSController` and `CVSCheckout`, respectively.

The subsequent exception handling actions as shown in Figure 6.2 (b.2) and (b.3) are similar to those in the first scenario except that the interface evolution problem is avoided by prepaying an exception design cost in the TTM iterations.

6.1.3 The third scenario

The third scenario makes use of unchecked exception exclusively, including `JavaCVS`. Figure 6.2 (c.1) depicts the situation. You can see that although throwing a new unchecked exception does not require changing component interfaces, the exception hole becomes hidden. How does `CVSCheckout` know that it has to catch the unchecked `ResponseException` thrown by `JavaCVS`? If you are lucky, the developers of `JavaCVS` will have written such information in the API documentation and, by chance, you have read it. If they did not, eventually testers or customers may notify you when they encounter the failure. Anyway, hopefully when you know the problem you can fix it; see Figure 6.2 (c.2). Finally, you reach the same robustness level as in the two pervious scenarios; see Figure 6.2 (c.3).

6.1.4 Cost analysis of the example

In all of the three scenarios, we have shown the usefulness of the robustness model from three primary aspects: (1) explicitly defining and documenting robustness requirement for a component, (2) guiding exception handling design, and (3) supporting exception handling in a staged manner. To evaluation the distribution of the exception handling cost in these scenarios, we further investigated the primitive operations involved in implementing CVSCheckout and CVSController, as shown in the third and fourth columns of Table 6.1. We then assigned numerical value to the primitive operations and calculated the total cost involved in all iterations of the three scenarios. The cost value and its percentage over the total cost of each scenario are shown in the fifth and sixth columns of Table 6.1, respectively. Table 6.2 lists the cost value assigned to each primitive operation. The cost distribution is illustrated in Figure 6.3 (a), including all costs of the listed primitive operations in Table 6.1, and Figure 6.3 (b), excluding the cost of error-handling, cleanup, selection, and alternative. There are a few observations that can be of interest.

Table 6.1: Cost analysis of the three scenarios

Scenario	Iter.	Primitive operation in CVSCheckout	Primitive operation in CVSController	Cost value	%
First	1	wrapping, signaling, catching _c		3	4.8
	2	declaring _c , wrapping, error-handling, cleanup,	catching _c , wrapping, signaling	29	46.0
	3		declaring _c , wrapping, selection, alternative,	31	49.2
Second	1	declaring _{c,pre} , catching _c , wrapping, signaling	declaring _{c,pre} , catching _c , wrapping, signaling	26	36.6
	2	error-handling, cleanup		20	28.2
	3		selection, alternative	25	35.2
Third	1			0	0
	2	delcaring _u , catching _u , wrapping, signaling, error-handling, cleanup		32	46.4
	3		declaring _u , catching _u , wrapping, signaling, selection, alternative	37	53.6

Table 6.2: The cost value assigned to each primitive operation

Primitive operations	Assigned cost value	Note
wrapping	1	Wrapping is supported by the Java language.
signaling	1	Signaling is supported by the Java language.
catching _c	1	Catching a checked exception is reminded by the Java compiler.
catching _u	5	Catching an unchecked exception requires manual effort of the programmers.
declaring _u	5	Declaring an unchecked (on demand) requires some design effort.
declaring _c	5	Declaring a checked (on demand) requires some design effort.
declaring _{c,pre}	10	Declaring a checked up-front requires even more design effort.
cleanup	5	Although Java provides the finally clause to release resources, invoking resource disposal methods require some effort.
selection	5	Designing the selection mechanism requires some effort.
error-handling	15	The error-handling involves making a checkpoint, restoring/dropping the checkpoint.
alternative	20	Designing and implementing an alternative is a non-trivial work.

6.1.4.1 Summary

- All of the three scenarios eventually achieve the same robustness level. That is, *the use of checked and unchecked exceptions is independent of robustness*. In other words, once a robustness goal has been defined, checked and unchecked exceptions can achieve the same goal. Although developers have been taught

that checked exceptions are superior to unchecked exceptions, or vice versa, in practice, taking any position does not help build robust applications. We believe that it is the lack of robustness goals that hampers exception handling design and implementation. This research proposed a robustness model with four well-defined goals for exception handling design. As a result, the use of checked and unchecked exceptions remains an implementation decision.

- Consider the primitive operations of the second iteration of the first scenario in Table 6.1. The declaring_c yields an interface change of CVSCheckout, which affects all of its callers. CVSController reflects such a change by utilizing the primitive operations of catching_c, wrapping, and signaling. Thus, we can conclude that the total cost of interface evolution cause by a component is:

$$\text{cost}(\text{declaring}_c) + N * [\text{cost}(\text{catching}_c) + \text{cost}(\text{wrapping}) + \text{cost}(\text{signaling})],$$

where N is the number of components calling the component changing its interface. Recursively applying, we can calculate the ripple effect cost of interface evolution.

- Theoretically, it is possible to calculate that which scenario is better when we add a new method or class to an existing system. Nevertheless, it would be impractical in general. For example, sometimes you just do not know how and where to design the exceptions. In addition, you may need some tangible and measurable criteria to engage to design with exceptions up front. Nevertheless, the baseline is that if your application components are not intended to be used as a third-party component by other developers or other projects, the first and third scenario is more flexible than the second. In this situation, cost of the interface

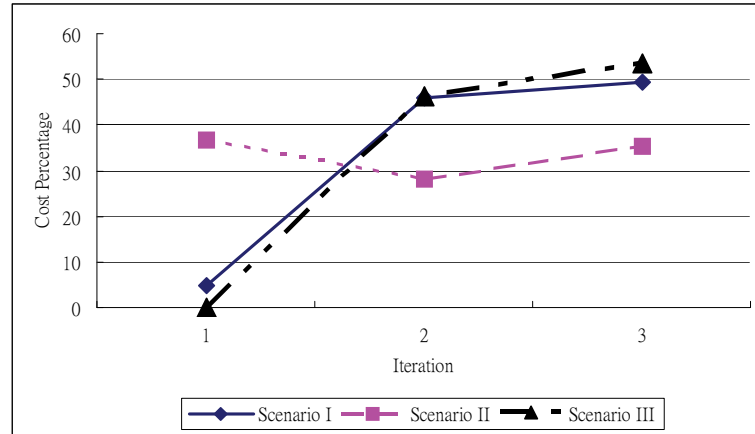
evaluation can be regarded as a necessary rework especially when you apply iterative and incremental design methods such as agile methods.

6.1.4.2 Characteristics of the first scenario

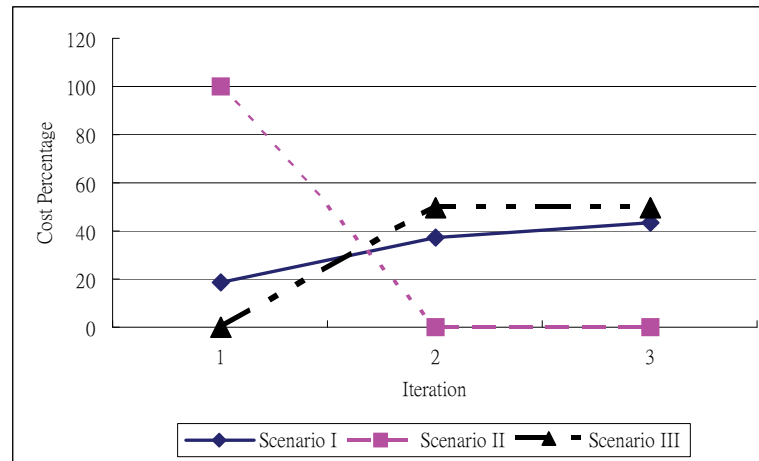
- By properly applying exception handling best practices, exception handling design can be flexible even if checked exceptions are used. For example, from Figure 6.3 (a), you can see that the cost curves in the first and the third scenarios are similar, except that in the TTM iteration of the first scenario there is a small initial cost in catching the checked exception, wrapping it to unchecked one, and signaling it. Such initial cost is similar to implement the *Replace Ignored Checked Exception with Unchecked Exception* refactoring.
- As shown in Figure 6.3 (b), exception design cost is incrementally paid, which is a good one; however, if it is not properly managed, the application could probably be massed up.

6.1.4.3 Characteristics of the second scenario

- The up front design avoids the interface evolution problem. Note that this does not mean that you have come up with any sophisticated exception handling design yet. By wrapping a checked exception to the unchecked `UnhandledException`, you effectively delay the design and implementation cost of error-handling, cleanup, selection, and alternative to meet the TTM requirement.
- When checked exceptions are designed up front, the application's exception handling flows are firmed since there is no interface evolution cost in the robustness iterations in the second scenario. This situation becomes extremely clear in Figure 6.3 (b) where we do not consider the common exception



(a)



(b)

Figure 6.3: Cost distribution curves of the three scenarios: (a) including all exception handling cost and (b) excluding the cost of error-handling, cleanup, selection, and alternative

handling cost in robustness enhancement; that is, the cost of error-handling, cleanup, selection, and alternative.

- Increasing the cost of error-handling, cleanup, selection, and alternative will decrease the percentage of the exception handling cost (i.e., the up front design

of checked exceptions) and as a result produces a cost distribution curve similar to the first and the third scenarios. shows such an example where the cost value of error-handling, cleanup, selection, and alternative is multiplied ten times. From Figure 6.3 and , we can conclude that using checked exceptions exclusively is not justifiable if the ultimate goal is G1.

6.1.4.4 Characteristics of the third scenario

- Unchecked exceptions fully support incremental design of exceptions.
- However, unchecked exceptions create implementation dependency and the compiler support is no longer available. For example, suppose that a new unchecked exception `JavaCVSException` is being substituted for `ResponseException`. In this case, the exception handler in `CVSCheckout` is no longer useful because it cannot be bound to the new exception.

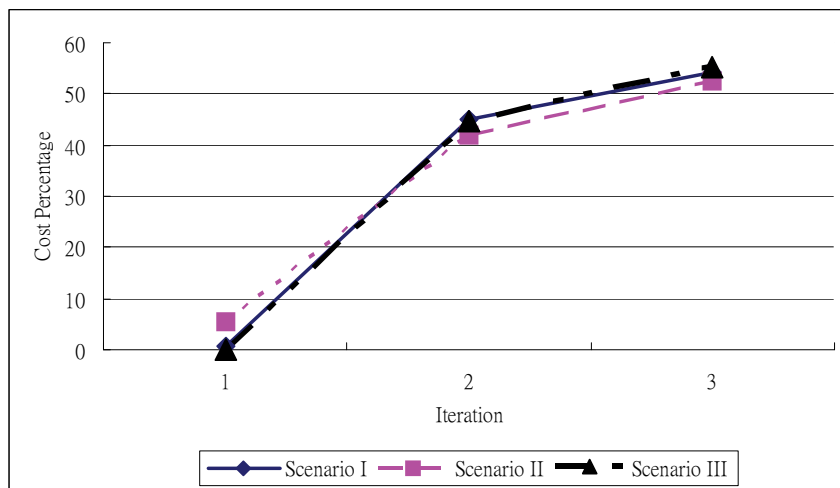


Figure 6.4: Cost distribution covers after increasing the cost of error-handling, cleanup, selection, and alternative ten times

6.2 A Case Study of Exception Handling Refactoring

We conducted a case study by applying the exception handling refactorings presented in Chapter 4 on a credit scoring system (CSS) to assess its usefulness¹. In this section, the CSS and the steps of applying EH refactorings are introduced and some field data to reflect the cost and effectiveness of the proposed approach are presented.

6.2.1 The credit scoring system

CSS is a widely used banking application in Taiwan that has been deployed since December 2003. A bank clerk checks a customer's credit history with CSS in processing his/her loan application. CSS forwards the request to an information portal hosted by the Joint Credit Information Center (JCIC), which is the sole credit reporting agency in Taiwan that collects credit data from financial institutions. Upon receiving replies from the JCIC portal, CSS retrieves the result and directs it to the requesting clerk.

CSS is a web application developed in Java with the Java server page (JSP) technology. Access to CSS is authenticated and authorized through a Domino server. CSS relies on IBM Message Queue to forward user queries to the JCIC portal. Once a query is processed, JCIC portal puts the result back to Message Queue. Lastly, CSS gets the result and sends it to the users. Figure 6.5 illustrates the system context diagram of CSS. Note that in Figure 6.5 the Domino server is a shared component that is used by other banking applications as well.

¹ The case study was conducted with the help of I-Lang Wu [50].

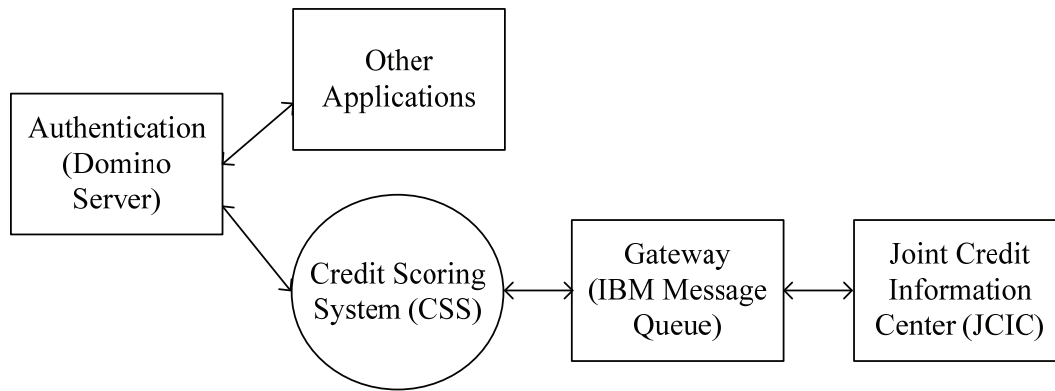


Figure 6.5: System context diagram of the credit scoring system (CSS)

The developers of CSS used exceptions to represent errors reported by JSP and other standard Java APIs. However, they were also forced to work with return code, which is Message Queue’s way of representing errors. At the time CSS was developed, exception handling was not a primary issue and the design decisions were left with the programmers. As a result, the use of exceptions and return code were mixed in an arbitrary manner; many checked exceptions were caught and ignored; unchecked exceptions were not caught which led to an unexpected termination of the JSP program; and failure atomicity of each component was not investigated. According to our definition, all CSS components are of G0.

6.2.2 Strategies to apply the refactorings

What CSS needs is the EH refactorings introduced in Chapter 4. Since EH refactorings are big refactorings that take time and consume human resources, good reasons must be given to convince the management. A reason such as “exception handling was badly designed before and we need to investigate the overall system to improve the design” is probably true but seemingly a bad one. You may be asked: “Why did you deliver poor quality software to customers in the first place?” In addition, because investigating the overall system requires significant initial efforts before the

quality improvement results become measurable, it can be difficult to convince the management to support the refactoring activity.

In our experience, we found that bug fixing is a good way to start big refactorings for two reasons. First, bug fixing is a strong force that draws the developer's attention to deal with exceptions. For example, known bugs must be fixed before software release and user-reported bugs must be fixed to prevent further complaints. Second, cost-effectiveness measurements can be provided to convince the management to support the refactoring because fixing bugs certainly improves software quality, saves maintenance cost, and increases customer satisfaction. In this case study, we collected bugs reported by a bank customer in the year of 2005 and identified top three most frequently occurred bugs attributed to poor exception handling. The identified bugs are shown in Table 6.3. Note that the count of login failures collected on the Domino server could be contributed by the applications other than CSS as well.

In this case study, all components in the call chain that may have caused the three bugs were identified via code review. Then, for each component in the call chain, we sought for EH smells. If a component had no EH smell, another workflow (not covered in this dissertation) to identify normal logic problems was conducted. Alternatively, if EH smells were discovered, we further set an exception handling goal of the component based on the robustness level obtained by consulting with the customer. Next, we wrote a test to reveal the bug, applied refactoring, and ran the test to check whether the bug was fixed. We repeated the last two steps until the test passed.

Table 6.3: Statistics of the top three most frequently occurred bugs attributed to exception handling in the year of 2005

Reported failure	Failing operation/ possible reason	Failure count	Current restoration actions	Restoration time per failure
Cannot make a connection to Domino server	User login/ Connections of Domino Internet Inter-ORB Protocol (DIIOP) were exhausted. It was likely that DIIOP connections were not correctly disposed.	125	1. Rebooting Domino server 2. Rebooting CSS	About 20 minutes
MQ server connection is broken	Sending request to JCJS/ MQ connections were exhausted. Perhaps they were not properly released.	66	1. Rebooting MQ Server 2. Rebooting CSS	About 10 minutes
Cannot send messages to MQ server	Sending request to JCJS/ There were two possible reasons: 1. Message queue was full. 2. Gateway was offline due to maintenance or crash.	10	1. Rebooting MQ Server 2. Rebooting CSS	About 10 minutes

6.2.3 EH refactoring in action

In what follows, we present the detail steps of refactoring to enhance the robustness of the “user login operation”. The refactoring also fixed the bug of “cannot make a connection to Domino server”. Since the bank customer had two replicated Domino servers, they required that if connection to the primary server fails, the system automatically connects to the secondary server. Thus, the refactoring goal of the login operation was set to G3, which was achieved incrementally as described next.

6.2.3.1 From G0 to G1

The `check` method of the `LoginChecker` class implements the login operation. Figure 6.6 shows the source code of the method, which accepts two parameters, a user id and a password, for authentication. If the user id and the password are valid, the invocation of the static method `createSession` on `NotesFactory` class returns a session object. Otherwise, an `AuthException` is raised. The `NotesFactory` class also raises an `AuthException` if it runs out of available session objects. Session objects are shared resources that must be released in the `finally` clause.

Initially, the method is G0. As can be seen, the `check` method employs return code (i.e., a Boolean value) to report errors. Thus, we applied *Replace Error Code with Exception* refactoring to achieve G1. Figure 6.7 is the refactored program where the modified code is underlined. The `check` method declares a new checked `AuthenticationException` as a means to report authentication failure. Note that we do not directly propagate the `AuthException` because it is an implementation-dependent exception that should be hidden from the callers of the `check` method.

```

public boolean check(String userId, String password) {
    Session session = null;          boolean result = false;

    try{
        session = NotesFactory.createSession(mServer, userId, password);
        result = true; }

    catch (AuthException e) {
        e.printStackTrace();          result = false; }

    finally {
        if (session != null) {
            try { session.recycle(); }

            catch (NotesException ex) { Log(ex); }

        }
    }

    return result;
}

```

Figure 6.6: The check method before refactoring

```

public void check(String userId, String password) throws AuthenticationException
{
    Session session = null;

    try{ session = NotesFactory.createSession(mServer, userId, password); }

    catch (AuthException e) { throw new AuthenticationException (e); }

    finally { /* code for cleanup */ }
}

```

Figure 6.7: The check method after refactoring for G1

6.2.3.2 From G1 to G2

To achieve G2, check's failure atomicity must be guaranteed. Although the original implementation of the `finally` clause, shown in Figure 6.6, has correctly released the session object by invoking its `recycle` method, it does so with a nested `try` block.

Thus, we applied *Replace Nested Try Block with Method* to remove the smell by extracting code from the `finally` clause to a newly created `recycle` method, which accepts a session object as a parameter:

```
finally { recycle(session); }
```

6.2.3.3 From G2 to G3

As mentioned in Section 6.2.1, the authentication server was shared by several banking applications. Thus, incorrect release of session objects by other applications may exhaust the resource and cause the `check` method to fail. In this situation, the users hoped that CSS could try to connect to the secondary authentication server.

We applied *Introduce Resourceful Try Clause* to achieve G3 and to avoid the spare handler smell. The resulting program is listed in Figure 6.8.

6.2.4 Results

We fixed the three failures presented in Table 6.3 with the proposed refactorings and collected failures reported by the customer from September 2006 to December 2006. As illustrated in Table 6.4, compared to the number of failures reported in the same period of time in 2005, we found that the robustness of CSS has been significantly improved. Further explanations of the exceptions that remain in Table 6.4 are provided as follows:

```

public void check(String userId, String password) throws AuthenticationException {

    Session session = null;
    int maxAttempt = 1;
    int attempt = 0;
    boolean retry = false;

    do {
        try{    retry = false;
            if (attempt == 0)
                session = NotesFactory.createSession(primary, userId, password);
            else
                session = NotesFactory.createSession(secondary, userId, password);
        }
        catch (AuthException e) {
            attempt++;
            retry = true;
            if (attempt > maxAttempt)
                throw new AuthenticationException (e);
        }
        finally {
            recycle(session);
        }
    } while (attempt<= maxAttempt && retry)
}

```

Figure 6.8: The refactored check method for G3

- In September 2006, we still received several failure reports of “Cannot make a connection to Domino server”. We then discovered that these failures were caused by other applications that did not properly release the connection resources. Thus, after discussion with the customer, code reviews were ordered for all applications that access the Domino servers to find EH smells and to remove identified smells by applying the proposed refactorings. The entire process took two months. The

result was a success and no such failure was reported after November 2006.

- Three failures of “MQ server connection broken” were reported in September 2006. These failures, however, should not be counted because they were not caused by exception handling bugs in CSS. At that time the number of users from bank branches was significantly increased due to the acquisition of another bank. Consequently, concurrent users exceeded the maximum value of the original configuration, which was increased in the due course.
- Similar to the pervious reason, we received four failure reports of “Cannot send messages to MQ server.” These failures should not be counted, either. In particular, since the refactored operation of sending messages to the Message Queue server became G3, retries were automatically conducted that reduced the number of reported failures. We investigated the log file of the Message Queue server and found that the reported failures were fewer than what were actual logged.

Table 6.4: The number of failures reported before and after refactoring

Reported failure	Failure count from Sep. 2005 to Dec. 2005				Failure count from Sep. 2006 to Dec. 2006			
	Sep.	Oct.	Nov.	Dec.	Sep.	Oct.	Nov.	Dec.
Cannot make a connection to Domino server	13	5	9	14	10	5	0	0
MQ server connection broken	6	1	4	8	3/0	0	1/0	0
Cannot send messages to MQ server	0	0	0	2	3/0	1/0	0	0

Table 6.5: Code analysis of the refactoring case study

Code type	Total vs. modified code	Modification percentage
Non-commented line of code	14150 / 371	2.63%
Catch clauses	855 / 21	2.46%

Table 6.6: Time spent in the refactoring case study

Task	Man-hours	Sum
Collecting bugs reports in 2005	18 man-hours	18 man-hours (43.9%)
Refactoring cost (Code review)	7 man-hours (30.4%)	
Refactoring cost (Exception handling goal analysis)	4 man-hours (17.4%)	
Refactoring cost (Coding)	3 man-hours (13%)	
Refactoring cost (Testing)	9 man-hours (39.2%)	
Total man-hours in applying refactoring		23 man-hours (56.1%)
Total man-hours in improving robustness		41 man-hours

Table 6.7: Cost-benefit analysis in terms of money saved

Cost element	Cost calculation
Maintenance cost in 2005	$201 \text{ (failure)} * 2 \text{ (hours)} * 100 \text{ (US\$)} = 40,200 \text{ (US\$)}$
Refactoring cost	$41 \text{ (man-hours)} * 100 \text{ (US\$)} = 4,100 \text{ (US\$)}$
Maintenance cost saving in the following year (money)	$40,200 \text{ (US\$)} - 4,100 \text{ (US\$)} = 36,100 \text{ (US\$)}$
Maintenance cost saving in the following year (percentage)	$36,100 \text{ (US\$)} / 40,200 \text{ (US\$)} * 100 = 89.8\%$

6.2.5 Cost-effectiveness analysis

In our case study, we found that applying the proposed refactorings is cost-effective and convincing. Table 6.5, Table 6.6 and Table 6.7 present some field data collected from the case study.

6.2.5.1 Code analysis

As shown in Table 6.5, the non-commented line of code of CSS was 14150. The modified line of code was 371 (2.62%). There were 855 catch clauses in CSS. We modified 21 of them (2.46%).

6.2.5.2 Time analysis

Table 6.6 shows time spent in the effort. A total of 18 man-hours were spent to review the bug reports of 2005. The top three most frequently occurred bugs caused by improper exception handling were identified. An additional 23 man-hours were spent to

apply the refactorings to fix the three bugs, including 7 man-hours in code review (30.4%), 4 man-hours in exception handling goal analysis (17.4%), 3 man-hours in coding (13%), and 9 man-hours in testing (39.2%). Thus, the refactoring effort cost 41 man-hours.

6.2.5.3 Cost analysis

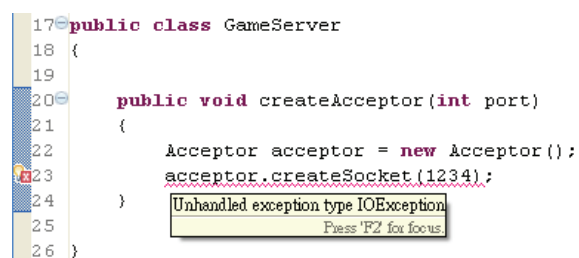
Table 6.7 shows a refactoring cost-benefit analysis in terms of money saved. In the year of 2005, a total of 201 failures reported were attributed to the top three bugs, see Table 6.3. From historical data, each instance of failure took approximately 2 hours of maintenance work, which resulted in a total of 402 hours. At the cost of 100 US\$ per man-hour, the total cost of handling summed to 40,200 US\$, which was registered as extra cost of the project. Using the same cost per man-hour, the cost of performing refactoring was 4,100 US\$. Therefore, fixing the three bugs probably saved 36,100 US\$ (89.8%) of maintenance cost in the following year.

6.3 An Exception Handling Refactoring Tool

Based on the concepts of the robustness model and exception handling refactorings presented in this dissertation, the author and his colleagues have developed a Eclipse plug-in for exception handling refactoring [47][48]. In this dissertation, we briefly present how the robustness model facilitates the development of a refactoring tool for exception handling.

6.3.1 The Eclipse Quick Fix for unhandled checked exceptions

The Eclipse platform provides a number quick fix features to keep the development work conducted in Eclipse smooth. One of the quick fixes deals with unhandled checked exceptions. Since Java deals with checked exceptions through the handle-or-declare rule, the call to a method that is specified to throw a checked exception must be surrounded by a `try` block or else have the exception declared in the signature of the caller. To smooth things up, the Eclipse Quick Fix provides two proposals on seeing a checked exception that violates the handle-or-declare rule: “Add throws declaration” and “Surround with try/catch”. The quick fix for checked exception is illustrated in Figure 6.9. In line 23, `acceptor.createSocket(1234)` throws a checked `IOException`. Since the developer has yet to enclose the call in a `try` block or have the same exception declared in the caller’s interface, a wavy red line is placed under the violating statement and a marker is placed on the left margin. When the mouse cursor is moved onto the marker, the violation is displayed. If the quick fix feature is invoked, the two proposals are shown, as in Figure 6.10.



```
17 public class GameServer
18 {
19
20     public void createAcceptor(int port)
21     {
22         Acceptor acceptor = new Acceptor();
23         acceptor.createSocket(1234);
24     }
25
26 }
```

Unhandled exception type IOException
Press 'F2' for focus.

Figure 6.9: The statement throwing an unhandled exception is underlined

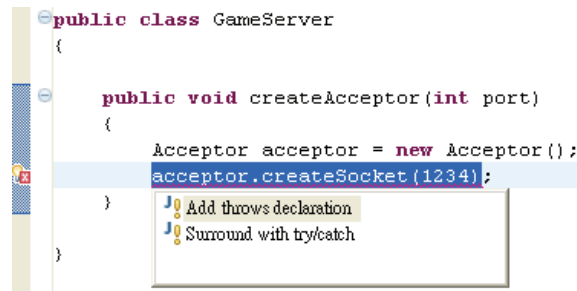


Figure 6.10: Exception handling proposals provided by Eclipse

```

public class GameServer
{
    public void createAcceptor(int port)
    {
        Acceptor acceptor = new Acceptor();
        try {
            acceptor.createSocket(1234);
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

```

Figure 6.11: Code after applying the “Surround with try/catch” proposal

Once a proposal is selected, code template defined in the proposal is inserted in the right place. For example, by selecting “Surround with try/catch”, the violating statement is fixed as in Figure 6.11 by the surrounding `try` block. To remind the developer that the exception handling work remains to be completed, Quick Fix places a comment “// TODO” inside the catch clause. In Eclipse terms, this is called a task and a list of all tasks can be browsed in the task view. Note that, however, while the developer is reminded of the task, no information is available on how the task might be completed.

Although the quick fix notifies the developer of the unhandled checked exceptions at the first opportunity, utilized without careful consideration, it can do more harm than help. Specifically, since the code added in the quick fix now satisfies the “handle-or-declare” rule, the Java compiler would no longer complain about the checked exception even if the exception handling is left blank. If the developer never

returns to take care of the task, the checked exception is equivalent to been ignored. The alternative is no better: choosing the proposal to declare the exception contaminates the caller's interface with a mechanism exception without really solving the problem.

6.3.2 Tool design

One way to help developers avoiding the afore-mentioned pitfalls is to include code templates that are consistent with the robustness model. By being consistent we mean the inserted code must at least fulfill the basic requirements. For example, if G3 is chosen and the exception handling policy is to retry, the inserted code should be able to put retry in action upon an exception. To do this, we may need to transform as well as insert code.

Figure 6.12 illustrates what can be done to provide the required functionality. A number of policies are designed subject to the robustness models. The models and the policies are made part of the Eclipse plug-in through Java annotations and collaborating classes in the utility libraries. As mentioned in Chapter 3.4, a component's exception handling goal is documented with Java annotations. These annotations are read by the tool for retrieving the exception handling policies (i.e., proposals) available under the specific goal. In particular, the policies are made into code templates as quick fix proposals. Note that different policies are made available according to the model chosen. For example, in G1, only two proposals are available for wrapping checked exceptions and for catching the unchecked exception at the top of the call chain. Note that although further customization can be performed on the inserted code template, we make it a requirement that the inserted code should be consistent with the requirement of the robustness model. This makes sure that inserted code is at least self-sufficient. Figure 6.13 shows a code template when a policy of "Add top-level catcher" (i.e., a big outer `try` block) is chosen.

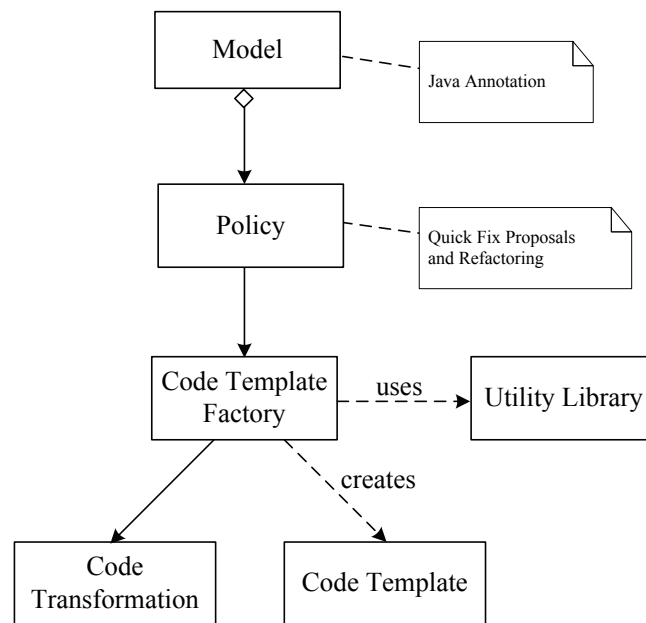


Figure 6.12: Conceptual model for the exception handling refactoring tool

<pre> acceptor.createSocket(1234); Add ExceptionChecker Add top-level catcher Retry with alternatives Retry with default values Retry with original values Add throws declaration Surround with try/catch </pre>	<pre> public void createAcceptor(int port){ Acceptor acceptor=new Acceptor(); try { acceptor.createSocket(1234); } catch (Exception e) { e.printStackTrace(); EHCore.getUncaughtExceptionHandler().handle(e); } } </pre>
--	---

Figure 6.13: The code template when choosing “Add top-level catcher” policy

6.4 A Robustness Visualization Tool

A robustness analysis tool based on static call chains analysis has been developed by the author's colleagues [49]. The tool also makes use of the Java annotations presented in this dissertation to document robustness level of components. The tool visualizes the collaboration of components in a call chain with a sequence diagram by using the robustness information documented in the annotations; see Figure 6.14.

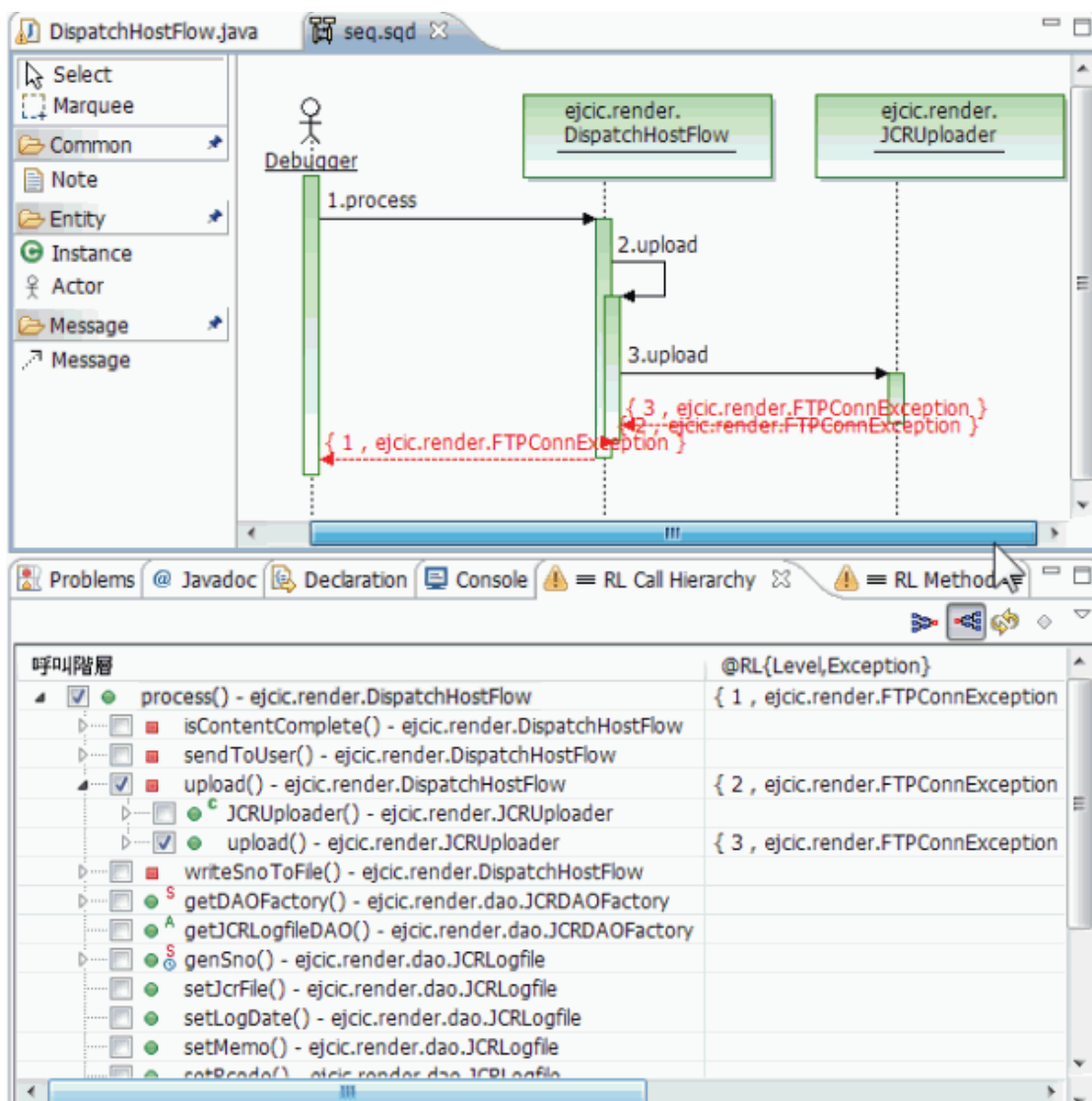


Figure 6.14: A sequence diagram with robustness information

Chapter 7 RELATED WORK

7.1 The Robustness Model of Exception Handling

The proposed robustness model of exception handling was designed to reflect exception handling capability of existing software and of failure atomicity. Achieving failure atomicity is a basic requirement for building robust systems. Researchers have used different terminologies to represent failure atomicity: in database and distributed computing, the “all-or-nothing” property embraces the concept of failure atomicity [51]; In Cristian’s research, “weakly tolerant” and “strongly tolerant” were used [17]; Meyer used “organized panic” and “retrying” [18]; Wirfs-Brock said “designing for recovery” [19]; Lee and Anderson used “idealized fault-tolerant component (IFTC)” to describe the behavior of a component with failure atomicity attribute [6]; In C++, a method is said to be “strong guarantee” if failure atomicity is implemented [13].

To denote real world applications that may not achieve failure atomicity, we added two levels to the robustness model. Goal level G0 declares a component the ground zero. Goal level G1 denotes a component taking the failing-fast approach [14] to deal with exceptions.

7.2 Exception Handling Refactoring

A variety of refactorings have been proposed, ranging from refactorings of object-oriented code to those of database, testing, pattern, and architecture-related code [29][52][53][54][55]. This dissertation moves refactoring research and practice towards the direction of robustness with exception handling.

EH smells and EH refactorings proposed in this dissertation are not invented from scratch. Actually, they are captured from common exception handling best practices and patterns. In particular, EH smells of *return code* and *ignored exception* are well-known

ones [10][20][29]. Although the names of EH smells of *unprotected main*, *dummy handler*, *nested try block*, and *spare handler* are new, they have been labeled as pitfalls to be avoided in exception handling [18][20][56][57].

The EH refactorings *Replace Ignored Checked Exception with Unchecked Exception* and *Replace Dummy Handler with Rethrow* basically reflect the failing-fast practice of exception handling [14]. Failing-fast suggests that clearly admitting your inability in exception handling as soon as possible increases robustness and code quality at least in the development and testing phases of software development. *Avoid Unexpected Termination with Big Outer Try Block* is inspired by patterns of *Big Outer Try Block* [56] and *Safety Net* [28]. *Replace Nested Try Block with Method* can be regarded as a derivative of *Extract Method* in the specific context of exception handling. Lastly, *Introduce Checkpoint Class* and *Introduce Resourceful Try Clause* reflect exception handling practices for guaranteeing failure atomicity [6][12][16][58].

Although exception handling and refactoring have been extensively studied individually, they are rarely considered together as a means to achieve a particular robustness goal. Two well-known exception handling refactorings proposed by Fowler, *Replace Error Code with Exception* and *Replace Exception with Test* [29] clearly improve code quality with respect to object-oriented design. However, compared to our refactorings, it is not clear how robust the refactored program will be by merely applying the two refactorings. In contrast, our refactorings achieve clearly defined exception handling goals.

Exception handling code is a popular target in the refactoring of object-oriented code to aspect-oriented code [59][60]. One such example is *Extract Exception Handling* [61] which moves a whole exception handling block (i.e., a `try` block in Java) to an aspect. Unlike our refactorings which are applicable to object-oriented programming,

this refactoring is applicable to aspect-oriented programming. In addition, *Extract Exception Handling* refactoring does not change robustness of the refactored program since it simply extracts a local exception handler to an aspect. Our refactorings, in contrast, focus on removing exception handling smells and can improve robustness and design quality.

7.3 Exception Handling Patterns

Several attempts have been made to apply patterns for exception handling. Our work does not intend to replace existing works, but to complement them. In this section we will briefly introduce these works and discuss the relationships between the proposed patterns and the existing ones.

In Haase's work, eleven Java idioms are presented which address best practices of exception handling. Eight of the eleven patterns, including *Expressive Exception Interface*, *Throwing Server*, *Checked Server Problem*, *Unchecked Client Problem*, *Homogeneous Exception*, *Exception Wrapping*, *Exception Tunneling*, and *Unhandled Exception*, reflect the best practices of using checked and unchecked exceptions. Two of them (i.e., *Smart Exception* and *Exception Hierarchy*) are related to the design of exception and exception hierarchy. The last one, *Safety Net*, is used to capture uncaught exceptions. Essentially, Haase's idioms can be used in almost any Java program. In the implementation of our pattern language, we directly applied *Exception Hierarchy* in *Exception Hierarchy for Abnormal Behavior*. The implementation of *Safety Method* and *Safety Command* is similar to *Safety Net*, though their contexts are different.

Seven language neutral patterns are presented in Longshaw and Woods's work [62]. Four patterns, including *Split Domain and Technical Errors*, *Make Exceptions Exceptional*, *Hide Technical Detail from Users*, and *Unique Error Identifier* are related

to exception design and usage. Two patterns, *Log at Distribution Boundary* and *Don't Log Domain Errors*, cover exception logging. One pattern, *Big Outer Try Block*, is similar to Haase's *Safety Net*.

Renzel's pattern language includes seven patterns [41]. Four patterns, including *Error Object*, *Exception Hierarchy*, *Exception Abstraction*, and *Exception Wrapper*, are related to exception class design. The remaining patterns are *Error Detection*, *Default Error Handling*, and *Backtrace*. It is interesting to compare the *Backtrace* pattern with the proposed patterns. Cited from the pattern, *Backtrace* solves the problem that "How do you collect and trace useful information that helps the system developers or the maintenance team analyze the error situation? Especially when we have no or limited access to the stack administered by the system itself." *Backtrace* further suggests making use of the call-stack information and writing the information to a log file. As a result, the log file reflects the history of the error. It is obvious that by applying our patterns the exception object can convey more accurate exception information which not only facilitates exception handling at runtime but also helps error inspection at maintenance.

In summary, existing exception handling pattern languages primarily address the issues of exception class design, usage, exception detection, and handling of uncaught exceptions. These issues are essential and can be applied in general software development contexts. Our patterns bridge a gap in exception handling practices in using exceptions to convey damage information. As a result, we avoid ignoring of subsequent exceptions and overwriting of declared exceptions.

Chapter 8 CONCLUSION

8.1 Contribution

In this dissertation, we have discussed Java exception handling design and implementation limitations and proposed approaches to improving software robustness with exception handling. The primary components of our approaches include a staged robustness model, exception handling refactorings, and patterns. We conducted case studies and developed tools to evaluate the usefulness and effectiveness of the proposed approaches. The major contributions of this dissertation are:

- establishing a staged robustness model with four clearly defined goals to simplify upfront exception handling design;
- capturing exception handling smells and refactorings to improve exception handling design after software is constructed;
- proposing patterns for solving the Java exception handling overridden problem;
- suggesting possible Java language enhancements for exception handling, including exception class design and idioms for exception throwing as well as exception collecting; and
- introducing a new way to apply Java annotations to enable exception handling tool support;

In addition, the proposed approach is applicable to checked and unchecked exceptions. Since a consensus on how to apply checked and unchecked exceptions is non-existent, having an approach to cover both categories of exceptions could be a significant advantage for Java developers to facilitate the development of robust

software.

8.2 Future Work

Our future work directions include:

- To apply the research to a variety of popular languages, including compiling languages such as C++, C#, VB.NET and scripting languages such as Groovy, JavaScript, Python, and TCL;
- To capture more exception handling smells and refactorings, and to develop tool support for automatic exception handling smells detection; and
- To extend the research to cover exception handling in software processes and architecture.

References

- [1] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language*, 4th ed., Addison-Wesley, 2006.
- [2] B. Venners, "Failure and exceptions: A conversation with James Gosling, Part II," Available from <http://www.artima.com/intv/solidP.html>, 2003.
- [3] B. Eckel, "Does Java need Checked Exceptions?" Available from <http://www.mindview.net/Etc/Discussions/CheckedExceptions>.
- [4] B. Venners and B. Eckel, "The trouble with checked exceptions: A conversation with Anders Hejlsberg, Part II," Available from <http://www.artima.com/intv/handcuffsP.html>, 2003.
- [5] B. Eckel, *Thinking in Java*, 4th ed., Prentice Hall, 2006, pp. 477-478.
- [6] P. A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice*, 2nd ed., Wien, 1990.
- [7] A. F. Garcia, C. M. F. Rubira, A. Romanovsky, and J. Xu, "A comparative study of exception handling mechanisms for building dependable object-oriented software," *Journal of Systems and Software*, vol. 59, no. 2: 197-222, 2001.
- [8] J. B. Goodenough, "Exception Handling: Issues and a Proposed Notation," *CACM*, vol. 18, no 12: 683-696, 1975.
- [9] J. L. Knudsen, "Exception Handling versus Fault Tolerance," *Proc. of ECOOP'00*, Springer-Verlag, 2001, pp. 1-17.
- [10] S. Stelting, *Robust Java: Exception Handling, Testing and Debugging*, Prentice Hall PTR, 2005.
- [11] R. Miller and A. Tripathi, "Issues with exception handling in object-oriented systems," *LNCS*, vol. 1241: 85-103, Springer, 1997.
- [12] J. Bloch, *Effective Java Programming Language Guide*, Addison Wesley, 2001.

- [13] B. Stroustrup, *The C++ Programming Language*, 3rd ed., Addison-Wesley, 2000.
- [14] J. Shore, "Fail fast," *IEEE Software*, 21 (5), 2004, pp. 21-25.
- [15] P. A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice*, 2nd ed., 1990, Springer.
- [16] L. L. Pullum, *Software Fault Tolerance Techniques and Implementation*, Artech House, 2001.
- [17] F. Cristian, "Exception handling and software fault tolerance," *IEEE Transactions on Computers*, c-31 (6), 1982, pp. 531-540.
- [18] B. Myer, *Object-Oriented Software Construction*, 2nd ed., Prentice Hall, 1997.
- [19] R. Wirfs-Brock, "Designing for recovery," *IEEE Software*, 23 (4), 2006, pp. 11-13.
- [20] A. Müller and G. Simmons, "Exception Handling: Common Problems and Best Practice with Java 1.4," *Net.ObjectDays 2002*, 2002, Last checked on 2008-6-12,
<<http://www.old.netobjectdays.org/node02/de/Conf/publish/slides.html>>.
- [21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [22] W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison Wesley, 1995, pp.98-99.
- [23] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2003.
- [24] T. Archer and A. Whitechapel, *Inside C#*, 2nd ed., Microsoft Press, 2002.
- [25] S. Northover and M. Wilson, *SWT: The Standard Widget Toolkit, Volume 1*, Addison-Wesley, 2004.
- [26] C. Szyperski, D. Gruntz, and S. Murer, *Component Software: Beyond Object-Oriented Programming*, 2nd ed., Addison-Wesley, 2002, pp.52-53.
- [27] M. Fowler, "Public versus Published Interfaces," *IEEE Software*, vol. 19, no. 2, Mar/Apr, 2002, pp.18-19.

- [28] A. Haase, "Java idioms: exception handling," *EuroPLoP'2002*, 2002.
- [29] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.
- [30] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transaction on Software Engineering*, 30 (2), 2004, pp. 126- 139.
- [31] B. V. Haecke, *JDBC 3.0: Java Database Connectivity*, M&T Books, 2002.
- [32] R. Wirfs-Brock, "Toward exception-handling best practices and patterns," *IEEE Software*, 23 (5), 2006, pp. 11-13.
- [33] R. H. Campbell and B. Randell, "Error recovery in asynchronous systems," *IEEE Transaction on Software Engineering*, SE-12 (8), 1986, pp. 811-826.
- [34] M. Parsian, *JDBC Recipes: A Problem-Solution Approach*, Apress, 2005.
- [35] J. Bloch and N. Gafter, *Java Puzzlers: Traps, Pitfalls, and Corner Cases*, Addison-Wesley, 2005.
- [36] K. Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, 1996.
- [37] E. Gamma and K. Beck, *Contributing to Eclipse: Principles, Patterns, and Plug-Ins*, Addison-Wesley, 2004.
- [38] R. Wirfs-Brock and A. McKean, *Object Design: Roles, Responsibilities, and Collaborations*, Addison-Wesley, 2003.
- [39] MSDN, .NET Framework Class Library: Exception.Data Property, Last checked on 2008-08-12, <[http://msdn.microsoft.com/en-us/library/system.exception.data\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/system.exception.data(VS.80).aspx)>.
- [40] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, 1 (1), 2004, pp. 11-33.
- [41] K. Renzel, "Error handling, a pattern language," 1997, Last checked on 2007-07-24, <<http://www.objectarchitects.de/arcus/cookbook/exhandling/index.htm>>.
- [42] D. Schwarz, "Avoiding checked exceptions," 2004, Last checked on 2007-07-24,

<http://www.oreillynet.com/onjava/blog/2004/09/avoiding_checked_exceptions.html>

- [43] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, *A Pattern Language*, Oxford University Press, 1977.
- [44] C. Alexander, *The Timeless Way of Building*, Oxford University Press, 1979.
- [45] C. Alexander, M. Silverstein, S. Angel, S. Ishikawa, and D. Abrams, *The Oregon experiment*, Oxford University Press, 1988.
- [46] K. Beck and R. Johnson, "Patterns generate architectures," *Lectures Notes in Computer Science*, vol. 821, 1994, pp. 139-149.
- [47] T.-C. She, C.-T. Chen, and C.-Y. Hsieh, "Realizing Exception Handling Strategy by Extending Eclipse Quick Fix Unhandled Exception," *Proceedings of the Second Taiwan Conference on Software Engineering*, 2006, pp.89-94. (in Chinese)
- [48] T.-C. She, "An Exception Handling Architecture and Utility Support for Java Language," Master thesis, National Taipei University of Technology, 2006.
- [49] Y.-F. Chen, "Tool Support for Implementing Robustness Models for Exception Handling," Master thesis, National Taipei University of Technology, 2008.
- [50] C.-T. Chen, Y. C. Cheng, C.-Y. Hsieh, I.-L. Wu, "Exception handling refactorings: Directed by goals and driven by bug fixing," *Journal of Systems and Software*, doi:10.1016/j.jss.2008.06.035, 2008.
- [51] A. S. Tanenbaum and M. V. Steen, *Distributed Systems: Principles and Paradigms*, Prentice Hall, 2002.
- [52] S. W. Ambler and P. J. Sadalage, *Refactoring Databases: Evolutionary Database Design*, Addison-Wesley, 2006.
- [53] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*, Addison-Wesley, 2007.
- [54] J. Kerievsky, *Refactoring to Patterns*, Addison-Wesley, 2006.
- [55] M. Lippert, and S. Roock, *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*, Wiley, 2006.

- [56] A. Longshaw, and E. Woods, "Patterns for the generation, handling and management of errors," *EuroPLOP'2004*, 2004.
- [57] W. Weimer and G. C. Necula, "Finding and preventing run-time error handling mistakes," *OOPSLA'04*, 2004, pp. 419-431.
- [58] R. J. Abbott, "Resourceful systems for fault tolerance, reliability and safety," *ACM Computing Surveys*, 22 (1), 1990, pp. 35-68.
- [59] D. Binkley, M. Ceccato, M., Harman, F. Ricca, and P. Tonella, "Tool-supported refactoring of existing object-oriented code into aspects," *IEEE Transactions on Software Engineering*, 32 (9), 2006, pp. 698-717.
- [60] M. P. Monteiro and J. M. Fernandes, "Towards a catalog of aspect-oriented refactorings," *Proc. Fourth Int'l Conf. Aspect-Oriented Software Development (AOSD)*, 2005, pp. 111-122.
- [61] R. Laddad, "Aspect-Oriented Refactoring, parts 1 and 2," The Server Side, 2003, Last checked on 2007-10-12. < <http://www.theserverside.com/>>.
- [62] A. Longshaw and E. Woods, "More patterns for the generation, handling and management of errors," *EuroPLOP'2005*, 2005.

VITA

Chien-Tsun Chen

Software Systems Lab.

Department of Computer Science and Information Engineering

National Taipei University of Technology (NTUT)

1, Sec. 3, Chung-Hsiao E. Rd., Taipei 106, Taiwan

E-mail: ctchen@ctchen.idv.tw

Tel: 886-2-27712171 ext. {4282}

Education

- PhD Dept. of Computer Science and Information Engineering, National Taipei University of Technology, Taiwan; Sep. 2008. (expected)
- Diploma Dept. of Electronic Engineering, National Taipei Institute of Technology, Taiwan; June 1994.

Experience

- Aug. 2006 - July 2009 Architect of the NSC project: “A New Approach to Supporting Design by Contract in the Java Language” at Software Systems Lab, NTUT, Taiwan.
- Aug. 2007 - July 2009 Architect of the NSC project: “Agile Exception Handling (I)” and “Agile Exception Handling (II)” at Software Systems Lab, NTUT, Taiwan.

Aug. 2006 - July 2008	Architect of the NSC project: “A Continuous Integration System for WiMAX Communication Software Development (I) & (II)” at Software Systems Lab, NTUT, Taiwan.
Aug. 2006 - July 2007	Architect of the NSC project: “A Study on a Front-End Regression Test Selection Tool for Agile Software Process” at Software Systems Lab, NTUT, Taiwan.
Aug. 2005 - July 2006	Architect of the NSC project: “A New Approach to Documenting Knowledge of Object-Oriented Software Systems: Using Multiple Aspects Unit Test” at Software Systems Lab, NTUT, Taiwan.
Feb. 2004 - Jan 2006	Teaching Assistant at Dept. of Computer Science and Information Engineering, NTUT, Taiwan.
Aug. 2004 - July 2005	Architect of the NSC project: “JCIS: An Open Source Continuous Integration System for Java Applications” at Software Systems Lab, NTUT, Taiwan.
Aug. 2004 - July 2005	Architect of the NSC project: “SyncFree2: The Enhancement of SyncFree Open Source Personal Data Synchronization Software” at Software Systems Lab, NTUT, Taiwan.
Aug. 2003 - July 2004	Architect of the NSC project: “SyncFree: An Open Source Personal Data Synchronization Software using Java Technology” at Software Systems Lab, NTUT, Taiwan.
May 2000 - Aug. 2002	Technical Director, CanThink Inc, Taiwan.
Oct 1996 - April. 2000	Programmer, CanThink Inc, Taiwan.

Research Interests

Exception Handling

Agile Methods

Pattern Languages

Software Architecture

Publications

Journal Articles:

- 1 C.-T. Chen, Y. C. Cheng, C.-Y. Hsieh, and T.-S. Hsu, "Delivering Specification-Based Learning Processes with Service-Oriented Architecture: A Process Translation Approach," (to appear) *Journal of Information Science and Engineering*, 2008. (SCI)
- 2 C.-T. Chen, Y. C. Cheng, and C.-Y. Hsieh, "Contract Specification in Java: Classification, Characterization, and a New Marker Method," (to appear) *IEICE Transactions on Information and Systems*, 2008. (SCI)
- 3 C.-T. Chen, Y. C. Cheng, C.-Y. Hsieh, and I.-L. Wu, "Exception Handling Refactorings: Directed by Goals and Driven by Bug Fixing," (to appear) *Journal of Systems and Software*, 2008. (DOI: 10.1016/j.jss.2008.06.035.) (SCI)

International Conference Papers:

- 1 Y. C. Cheng, P.-H. Ou, C.-T. Chen, and T.-S. Hsu, "A Distributed System for Continuous Integration with JINI," accepted by the 2008 International Conference on Distributed Multimedia Systems (DMS 2008), September 4-6, 2008, Boston, USA.

- 2 Y. C. Cheng, C.-T. Chen, and C.-Y. Hsieh, "ezContract: Using Marker Library and Bytecode Instrumentation to Support Design by Contract in Java," *Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC 07)*, December 5-7, 2007, Nagoya, Japan.
- 3 C.-T. Chen, Y. C. Cheng, and C.-Y. Hsieh, "Towards a Pattern Language Approach to Establishing Personal Authoring Environments in E-Learning," *Proceedings of the Sixth IASTED International Conference on Web-based Education (WBE 07)*, pp. 13-18, March 14-16, 2007, Chamonix, France.
- 4 C.-T. Chen, Y. C. Cheng, and C.-Y. Hsieh, "A Scenario-Based Approach for Modeling Abnormal Behaviors of Dependable Software Systems," *Proceedings of the 2006 International Computer Symposium (ICS 06)*, pp. 483-488, December 6-6, 2006, Taipei, Taiwan.
- 5 Y. C. Cheng, C.-T. Chen, and J. S. Jwo, "Exception Handling: An Architecture Model and Utility Support," *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC 05)*, pp. 359-366, December 15-17, 2005, Taipei, Taiwan.
- 6 M.-C. Chen, C.-T. Chen, Y. C. Cheng, and C.-Y. Hsieh, "On the Development and Implementation of a Sequencing Engine for IMS Learning Design Specification," *Proceedings of the 5th IEEE International Conference on Advanced Learning Technologies (ICALT 2005)*, July 5-8, 2005.
- 7 C.-T. Chen and Y. C. Cheng, "A Pattern Language for Personal Authoring in E-Learning," Published in the 11th Conference on Pattern Languages of Programs (PLoP2004), Sep. 8-12, 2004, Illinois, USA.
- 8 J. S. Jwo, H. C. Lu, and C.-T. Chen, "A Framework for Thin-Client-Server Computing," *Proceedings of the PAJava99*, 1999, London UK, pp. 125-135.

- 9 C.-T. Chen and J. S. Jwo, "World Wide Meet - Browsing the Web Not Alone," *Multimedia Information Systems in Practice*, edited by W. S. Chow, Springer-Verlag, 1998, pp. 549-556.
- 10 C.-T. Chen and J. S. Jwo, "A Java-Implemented Message-Routing Framework for World Wide Meet," *Proceedings of the 1998 International Computer Symposium (ICS 98)*, Taiwan, pp. 185-190.

Local Conference Papers:

- 1 C.-C. Hung, Chien-Tsun Chen, Y. C. Cheng, and C.-Y. Hsieh, "Reg4J2: A Debugging Tool based on Debugging History Management and Test Case Selection," *Proceedings of the Forth Taiwan Conference on Software Engineering*, Tainan City, June 13-14, 2008. (in Chinese)
- 2 T.-T. Kao, Chien-Tsun Chen, Y. C. Cheng, and C.-Y. Hsieh, "A Scrum Supporting Tool: Using Issue Tracking and Continuous Integration Systems," *Proceedings of the Forth Taiwan Conference on Software Engineering*, Tainan City, June 13-14, 2008. (in Chinese)
- 3 P.-H. Ou, Chien-Tsun Chen, Y. C. Cheng, and T.-S. Hsu, "JCIS2 : A Distributed Continuous Integration System," *Proceedings of the Forth Taiwan Conference on Software Engineering*, Tainan City, June 13-14, 2008. (in Chinese)
- 4 I-Lang Wu, Y. C. Cheng, Chien-Tsun Chen, and C.-Y. Hsieh, "Improving Robustness of Legacy Systems by Applying Exception Handling Models and Strategies," *Proceedings of the Third Taiwan Conference on Software Engineering*, Taichung City, June 8-9, 2007. (in Chinese)
- 5 C.-Y. Hsieh, W.-C. Lee, Chien-Tsun Chen, and Y. C. Cheng, "Reg4J: A Front-End Regression Test Selection Tool for Agile Process," *Proceedings of*

- the Third Taiwan Conference on Software Engineering*, Taichung City, June 8-9, 2007. (in Chinese)
- 6 Chia-Hao Wu, Tsui-Chen She, Chien-Tsun Chen, and Y. C. Cheng, "JCIS: Supporting Platform Dependent Builds," *Proceedings of the Third Taiwan Conference on Software Engineering*, Taichung City, June 8-9, 2007. (in Chinese)
 - 7 Tsui-Chen She, Chien-Tsun Chen, and C.-Y. Hsieh, "Realizing Exception Handling Strategy by Extending Eclipse Quick Fix Unhandled Exception," *Proceedings of the Second Taiwan Conference on Software Engineering*, pp.89-94, Taipei City, June 9-10, 2006. (in Chinese)
 - 8 C.-Y. Hsieh and Chien-Tsun Chen, "A Study on Applying JUnit Framework to Document Knowledge of Object-Oriented Software Systems," *Proceedings of the Second Taiwan Conference on Software Engineering*, pp.48-53, Taipei City, June 9-10, 2006. (in Chinese)
 - 9 Tien-Song Hsu, M.-C. Chen, C.-T. Chen, and Y. C. Cheng, "Realizing IMS Learning Design Using Business Process Execution Language and Web Services," *Proceeding of the First Taiwan Software Engineering Conference*, Taipei City, June 3-4, 2005. (in Chinese)
 - 10 C.-T. Chen, J.-Y. Lei, Y. C. Cheng, and C.-Y. Hsieh, "SyncFree: The Development of an Open Source Personal Data Synchronization Software," *Proceedings of Taiwan Area Network Conference 2004 (TANET2004)*, pp. 184-189, Oct. 27-29, 2004. (in Chinese)
 - 11 C.-T. Chen, T.-S. Shiu, M.-J. Chen, Y. C. Cheng, and C.-Y. Hsieh, "Using BPEL4WS to Describe IMS Learning Design," *Proceedings of Taiwan Area Network Conference 2004 (TANET2004)*, pp. 1135-1140, Oct. 27-29, 2004. (in

Chinese)

- 12 C.-T. Chen, W.-C. Lee, Y. C. Cheng, and C.-Y. Hsieh, "JavaCIS: A Continuous Integration System for JAVA Applications," Published in the 15th Workshop on Object-Oriented Technology and Applications (OOTA), Sep. 9, 2004. (in Chinese)
- 13 C.-T. Chen and Y. C. Cheng, "A Pattern Language for Web-Based Learning Authoring Environment," *Proceedings of Taiwan Area Network Conference 2003* (TANET2003), pp.365-371, Oct. 29-31, 2003. (in Chinese)

Certification

- 1 Certification of Completion, Intermediate Concepts of Capability Maturity Model Integration (CMMI), Nov. 4, 2005.
- 2 Certification of Completion, Introduction to Capability Maturity Model Integration (CMMI), Staged Representation, V1.1, Sep. 21-23, 2005.
- 3 Microsoft Certified Professional (MCP), Designing and Implementing Desktop Application with Microsoft Visual Basic 6.0, May 6, 2002.

Professional Society Memberships

Software Engineering Association of Taiwan (SEAT)