The Journal of Systems and Software 82 (2009) 333–345

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Exception handling refactorings: Directed by goals and driven by bug fixing

Chien-Tsun Chen[1], Yu Chin Cheng *, Chin-Yun Hsieh[2], I-Lang Wu

Department of Computer Science and Information Engineering, National Taipei University of Technology, 1, Sec 3, Chung-Hsiao E. Road, Taipei 106, Taiwan

ARTICLE INFO

Article history:
Received 23 January 2008
Received in revised form 23 June 2008
Accepted 24 June 2008
Available online 3 July 2008

Keywords:
Refactoring
Java exception handling
Object-oriented design

ABSTRACT

Exception handling design can improve robustness, which is an important quality attribute of software. However, exception handling design remains one of the less understood and considered parts in software development. In addition, 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. This paper applies refactoring to incrementally improve exception handling design. We first establish four exception handling goals to stage the refactoring actions. Next, we introduce exception handling smells that hinder the achievement of the goals and propose exception handling refactorings to eliminate the smells. We suggest exception handling refactoring is best driven by bug fixing because it provides measurable quality improvement results that explicitly reveal the benefits of refactoring. We conduct a case study with the proposed refactorings on a real world banking application and provide a cost-effectiveness analysis. The result shows that our approach can effectively improve exception handling design, enhance software robustness, and save maintenance cost. Our approach simplifies the process of applying big exception handling refactoring by dividing the process into clearly defined intermediate milestones that are easily exercised and verified. The approach can be applied in general software development and in legacy system maintenance.

1. Introduction

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" (Fowler, 2000), 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.

A variety of refactorings have been proposed, ranging from refactorings of object-oriented code (Fowler, 2000) to those of database, testing, patterns, and architecture-related code (Ambler and Sadalage, 2006; Meszaros, 2007; Kerievsky, 2006; Lippert and Roock, 2006). Regardless of which type of refactorings to perform, 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 (Fowler, 2000; Mens and Tourwé, 2004): (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 paper, we present *exception handling refactorings* (EH refactorings for short), 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

* Corresponding author. Tel.: +886 2 27712171x4232; fax: +886 2 87732945.
E-mail addresses: s1669021@ntut.edu.tw, ctchen@ctchen.idv.tw (C.-T. Chen), yccheng@ntut.edu.tw, yccheng@csie.ntut.edu.tw (Y.C. Cheng), hsieh@ntut.edu.tw, hsieh@csie.ntut.edu.tw (C.-Y. Hsieh), s3598026@ntut.edu.tw, chrisilwu@gmail.com (I-Lang Wu).
[1] Tel.: +886 2 27712171x4282.
[2] Tel.: +886 2 27712171x4231.

0164-1212/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2008.06.035

verify whether behavior remains unchanged (Mens and Tourwé, 2004). From an exception handling perspective, a system's behaviors can be categorized into the *normal part* and the *abnormal part* (Lee and Anderson, 1990). 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.

It is generally agreed that exception handling design is a complex problem and improving the design after the system is constructed is difficult. Following the general practice of solving a complex problem with divide and conquer, we believe that EH refactoring is best tackled by conducting a series of small refactorings. In other words, EH refactoring is a type of big refactorings. To this end, we identify four *staged exception handling goals* for guiding a big EH refactoring by dividing it into clearly defined intermediate milestones that are easily exercised and verified. We then identify *code smells* that impede the achievement of the goals and propose refactorings for their removal. Also, we suggest that the EH refactorings are best driven by fixing exception handling related *bugs* for two reasons. First, bugs constitute a strong force that draws the attention of the development team and provide a convincing argument to the management for conducting EH refactoring. Second, bug fixing provides a working context that makes the effort of EH refactoring accountable since the cost and benefit can be concretely measured. We conduct a case study involving a real world banking application that was actively in use when the proposed EH refactorings were conducted. The case study shows that, guided by staged goals and driven by bug fixing, exception handling design can be effectively improved with the proposed EH refactorings. A cost-benefit analysis in support of the claim is presented.

2. Goals of exception handling

Before attempting any refactoring to improve exception handling design, it is necessary to set a goal to direct the actions and to set the stopping criteria. Although our focus is on refactoring, the goals need not be different from those achieved with up front exception handling design. By surveying the literature, we have identified three different levels of robustness that are achieved by commonly adopted exception handling strategies; see Table 1. The three robustness levels, referred to as G1, G2 and G3 in Table 1, will serve as the goals for directing exception handling refactorings. An additional level, referred to as G0 in Table 1, is added to account for the status before refactoring.

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, and also known as.

Goal 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 3, there are many ways programmers can be led into creating components of robustness level at G0.

Goal 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 (Shore, 2004).

Goal 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 (Lee and Anderson, 1990; Myer, 1997; Pullum, 2001). 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 (Pullum, 2001). 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 (Cristian, 1982).

Goal 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.

Table 1
Robustness levels of a component and its effect on the program after encountering an exception

Element	Goal levels			
	G0	G1	G2	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	Strongly tolerant

Retry, design diversity, data diversity, and functional diversity are common methods to implement G3 (Lee and Anderson, 1990; Myer, 1997; Pullum, 2001; Wirfs-Brock, 2006). 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 (Cristian, 1982).

Note that G2 and G3 constitute *failure atomicity* (Cristian, 1982) and *designing for recovery* (Wirfs-Brock, 2006), 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 refactoring 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 Fig. 1.

2.1. 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 compilers make 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 (Stelting, 2005). 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 (Müller and Simmons, 2002). 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 compilers 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.

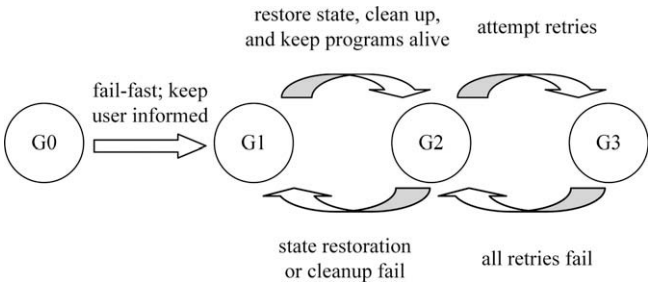

Fig. 1. Upgrading and degrading exception handling goals.

3. 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 (Fowler, 2000) and should be avoided in exception handling code as well (e.g., see the related work section for more information about removing duplicated exception handling code). In this paper, we identify six exception handling code smells, hereafter referred to as *EH smells*.

3.1. 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. A program is confusing and difficult to maintain when exceptions and return code are arbitrarily mixed.

3.2. Ignored checked exception

Ignored checked exceptions reduce robustness and make debugging difficult. A checked exception indicates the occurrence of an expected and recoverable error (Bloch, 2001; Stelting, 2005). 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.

3.3. 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.

3.4. Dummy handler

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

catch (AnException e) {e.printStackTrace();}, or
catch (AnException 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.

3.5. 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.

3.6. 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. Refactorings

The seven refactorings of Table 2 are applied to remove the EH smells and to achieve the proposed exception handling goals. The first one, *Replace Error Code with Exception*, is a well-known refactoring presented in (Fowler, 2000) 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 10 projects in the banking domain during the time span from spring 2005 to winter 2007. In Section 5, one such case is studied in detail.

Martin Fowler's format is used in presenting the refactorings (Fowler, 2000), which consists of five parts: name, summary, motivation, mechanics, and examples.

4.1. 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 */}
    }
```

4.1.1. 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 (Miller and Tripathi, 1997), 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.

4.1.2. 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.

4.1.3. 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

Table 2
Exception handling smells, refactorings, and goals

EH smell	Refactoring	Achieving goal
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

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.

4.2. 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 */}
}
```

4.2.1. 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.

4.2.2. 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.

4.2.3. 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 */}
}
```

4.3. 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"); }
```

4.3.1. 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 (Eckel, 2006).

4.3.2. 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. 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");
```

4.4. 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 */ }
}
```

4.4.1. 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:

- *Providing inline alternatives. A nested* `try` block is used to provide inline exception handler for a method call. Fig. 2 shows an example taken from (Haecke, 2002). 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.

Each of the use of the nested `try` block yields complicated program structures and easily results in a *Long Method* (Fowler, 2000).

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.

4.4.2. Mechanics

i. Apply *Extract Method* (Fowler, 2000) to each nested `try` block.
ii. Replace the nested `try` block with a call to the extracted method.
iii. Compile and test.

4.4.3. 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

```
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 (UnknowHostException ex) {   localHost = "localhost/127.0.0.1"; }
09        …
10       /* (2) update database  */
11       ...
12    } catch (SQLException e) { … }
13 }
```

Fig. 2. A nested `try` block providing an inline alternative.

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); }
```

4.5. 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(); }
}
```

4.5.1. 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.

4.5.2. 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.

4.5.3. Example

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) {/* con-
    structor */};
    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 Checkout-
    Exception(e);}
    finally {fcp.drop();}
}
```

4.6. 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)
```

4.6.1. 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 (Myer, 1997; Stelting, 2005; Wirfs-Brock, 2006). In a language that supports a retry model of exception handling (e.g., Eiffel), retrying is directly supported (Myer, 1997). 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 Fig. 2, 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.

4.6.2. 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.

4.6.3. 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 ReadUserEx-
            ception (e);}
        } while (attempt <= maxAttempt && retry)
}
```

5. A case study

We conducted a case study by applying the proposed approach on a credit scoring system (CSS) to assess its usefulness. In this section, we briefly introduce the CSS and the steps of applying EH refactorings. We further present some field data to reflect the cost and effectiveness of the proposed approach.

5.1. 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. Fig. 3 illustrates the

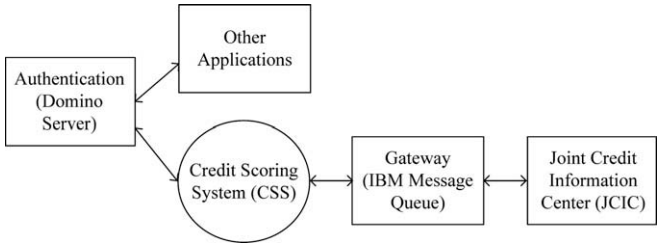

Fig. 3. System context diagram of the credit scoring system (CSS).

system context diagram of CSS. Note that in Fig. 3 the Domino server is a shared component that is used by other banking applications as well.

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.

5.2. Strategies to apply the refactorings

What CSS needs is the EH refactorings introduced in Section 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 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 paper) 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.

```
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;
}
```

Fig. 4. The check method before refactoring.

5.3. 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.

5.3.1. From G0 to G1

The `check` method of the `LoginChecker` class implements the login operation. Fig. 4 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

Table 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 min
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 min
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 min

```
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 */ }
}
```

Fig. 5. The check method after refactoring for G1.

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. Fig. 5 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.

5.3.2. From G1 to G2

To achieve G2, `check`'s failure atomicity must be guaranteed. Although the original implementation of the `finally` clause, shown in Fig. 4, 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); }
```

5.3.3. From G2 to G3

As mentioned in Section 5.1, 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.

```
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)
}
```

Fig. 6. The refactored check method for G3.

Table 4
A comparison of the number of failures reported before and after refactoring

Reported failure	Failure count from September 2005 to December 2005				Failure count from September 2006 to December 2006			
	September	October	November	December	September	October	November	December
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

We applied *Introduce Resourceful Try Clause* to achieve G3 and to avoid the *spare handler* smell. The resulting program is listed in Fig. 6.

5.4. Results

We fixed the three failures presented in Table 3 with the proposed refactorings and collected failures reported by the customer from September 2006 to December 2006. As illustrated in Table 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 4 are provided as follows:

- 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.

5.5. Cost-effectiveness analysis

In our case study, we found that applying the proposed refactorings is cost-effective and convincing. Tables 5–7 present some field data collected from the case study.

Table 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
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 7
Cost-benefit analysis in terms of money saved in the refactoring case study

Cost element	Cost calculation
Maintenance cost in 2005	201 (failure) × 2 (hours) × 100 (US$) = 40,200 (US$)
Refactoring cost	41 (man-hours) × 100 (US$) = 4100 (US$)
Maintenance cost saving in the following year (money)	40,200 (US$) − 4100 (US$) = 36,100 (US$)
Maintenance cost saving in the following year (percentage)	36,100 (US$)/40,200 (US$) × 100 = 89.8%

5.5.1. Code analysis

As shown in Table 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%).

5.5.2. Time analysis

Table 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.

5.5.3. Cost analysis

Table 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 3. From historical data, each instance of failure took approximately 2 h of maintenance work, which resulted in a total of 402 h. 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 4100 US$. Therefore, fixing the three bugs probably saved 36,100 US$ (89.8%) of maintenance cost in the following year.

5.6. Limitation and future work

It should be noted that the situation CSS faced is not uncommon in practice. Although exception handling has been extensively studied for decades, it is yet to become a first-class concern in most general purpose software development. Since exception handling design is a non-functional requirement, it is considered together with other competing and often conflicting project factors such as time-to-market, maintainability, and so on. Given limited project resource, tradeoffs are unavoidable and often with exception handling design ending up on the losing side. Thus, applying refactoring to improve exception handling design is in fact a practical approach towards robustness, especially since it can be performed in a cost-effective manner.

There are some limitations in our approach that could be improved in the future.

- *Architecture change*: In the case study, applying EH refactorings did not alter the application architecture. It is a fact that altering architecture after software is constructed is a very complicated issue of its own. Thus, if EH refactorings change the application architecture, the entire refactoring process would be much more complex. It should be an interesting research topic to investigate how EH refactoring relates to architecture refactoring, and vise versa.
- *Automatic EH smell detection*: The case study revealed EH smells via code review, which was conducted by the original developer of the application. For developers who are not familiar with the source code of the application under refactoring, it could be a great help if tool support of automatic EH smell detection is available.
- *Automatic EH refactoring*: We manually conducted EH refactorings in the case study. Although the big EH refactoring was staged according to the proposed exception handling goals, an automatic EH refactoring tool can further simplify the refactoring process. We envision the development of such an automatic EH refactoring tool because once the exception handling goal of a component has been specified, it is then possible to provide candidate refactoring proposals and to conduct automatic EH refactoring based on the chosen proposal.

6. Related work

EH smells and EH refactorings proposed in this paper 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 (Fowler, 2000; Müller and Simmons, 2002; Stelting, 2005). 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 (Longshaw and Woods, 2004; Müller and Simmons, 2002; Myer, 1997; Weimer and Necula, 2004).

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 (Shore, 2004). 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* (Longshaw and Woods, 2004) and *Safety Net* (Haase, 2002). *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 (Abbott, 1990; Bloch, 2001; Lee and Anderson, 1990; Pullum, 2001).

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* (Fowler, 2000), 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 (Binkley et al., 2006; Monteiro and Fernandes, 2005). One such example is *Extract Exception Handling* (Laddad, 2003), 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. Conclusion

In this paper, we first argued that exception handling refactoring differs from normal refactoring because code smells of exception handling usually cause bugs in code and EH refactoring can change the abnormal behavior of the refactored program as a means to improve robustness. We then presented the elements of the proposed approach, including exception handling goals, EH smells, and EH refactorings. Separately, each of the proposed elements seems to be straightforward. However, applying all of them together in a controlled manner is a non-trivial one. In this work, they are put together under a goal-directed and bug fixing process to incrementally improve robustness as well as design quality of a software system in a practical and manageable way. The case study demonstrates the usefulness and the cost-effectiveness of the proposed approach.

Acknowledgements

This work is funded by the National Science Council of Taiwan under the grants NSC96-2221-E-027-034 and NSC95-2221-E-027-046-MY3. We thank the anonymous reviewers for their constructive comments.

References

Abbott, R.J., 1990. Resourceful systems for fault tolerance, reliability and safety. ACM Computing Surveys 22 (1), 35–68.
Ambler, S.W., Sadalage, P.J., 2006. Refactoring Databases: Evolutionary Database Design. Addison-Wesley, New York.
Binkley, D., Ceccato, M., Harman, M., Ricca, F., Tonella, P., 2006. Tool-supported refactoring of existing object-oriented code into aspects. IEEE Transactions on Software Engineering 32 (9), 698–717.
Bloch, J., 2001. Effective Java Programming Language Guide. Addison-Wesley, New York.
Cristian, F., 1982. Exception handling and software fault tolerance. IEEE Transactions on Computers c-31 (6), 531–540.
Eckel, B., 2006. Thinking in Java, fourth ed. Prentice Hall PTR, New Jersey. pp. 477–478.
Fowler, M., 2000. Refactoring: Improving the Design of Existing Code. Addison-Wesley, New York.

Haecke, B.V., 2002. JDBC 3.0: Java Database Connectivity. M&T Books.
Haase, A., 2002. Java idioms: exception handling. In: EuroPLoP'2002.
Kerievsky, J., 2006. Refactoring to Patterns. Addison-Wesley, New York.
Laddad, R., 2003. Aspect-oriented refactoring, parts 1 and 2. The server side. <http://www.theserverside.com/>(accessed: 12.10.07).
Lee, P.A., Anderson, T., 1990. Fault Tolerance: Principles and Practice, second ed. Springer, New York.
Lippert, M., Roock, S., 2006. Refactoring in Large Software Projects: Performing Complex Restructurings Successfully. Wiley, New York.
Longshaw, A., Woods, E., 2004. Patterns for the generation, handling and management of errors. In: EuroPLOP'2004.
Mens, T., Tourwé, T., 2004. A survey of software refactoring. IEEE Transaction on Software Engineering 30 (2), 126–139.
Meszaros, G., 2007. xUnit Test Patterns: Refactoring Test Code. Addison-Wesley, New York.
Miller, R., Tripathi, A., 1997. Issues with exception handling in object-oriented systems. In: Aksit, M., Matsuoka, S. (Eds.), Advances in Exception Handling Techniques, LNCS 1241. Springer, New York, pp. 85–103.
Monteiro, M.P., Fernandes, J.M., 2005. Towards a catalog of aspect-oriented refactorings. In: Proceedings of the Fourth International Conference on Aspect-Oriented Software Development (AOSD). pp. 111–122.
Müller, A., Simmons, G., 2002. Exception handling: common problems and best practice with Java 1.4. Net.ObjectDays 2002. <http://www.old.netobjectdays.org/node02/de/Conf/publish/slides.html> (accessed: 12.06.08).
Myer, B., 1997. Object-Oriented Software Construction, second ed. Prentice Hall, New Jersey.
Pullum, L.L., 2001. Software Fault Tolerance Techniques and Implementation. Artech House, London.
Shore, J., 2004. Fail fast. IEEE Software 21 (5), 21–25.
Stelting, S., 2005. Robust Java: Exception Handling, Testing and Debugging. Prentice Hall, New Jersey.
Weimer, W., Necula, G.C., 2004. Finding and preventing run-time error handling mistakes. In: OOPSLA'04. pp. 419–431.
Wirfs-Brock, R., 2006. Designing for recovery. IEEE Software 23 (4), 11–13.

Chien-Tsun Chen is a PhD candidate at the Department of Computer Science and Information Engineering, the National Taipei University of Technology, Taiwan. His research interests include exception handling, pattern languages, and Design by Contract. Prior to commencing his graduate study, he served as the CTO of CanThink Inc., which provides e-learning software solutions and services. He is a member of the Software Engineering Association of Taiwan (SEAT).

Yu Chin Cheng is a professor at the Department of Computer Science and Information Engineering of the National Taipei University of Technology, Taiwan. His research interests include pattern languages for software design, software architecture and image analysis. He teaches object-oriented programming, object-oriented analysis and design, and software architecture. He received the MSE degree from the Johns Hopkins University in 1990 and the PhD degree from the University of Oklahoma, in 1993, both in Computer Science. He is a member of the IEEE Computer Society, the ACM, and the Software Engineering Association of Taiwan (SEAT).

Chin-Yun Hsieh is an associate professor at the Department of Computer Science and Information Engineering of the National Taipei University of Technology, Taiwan. He received his MS and PhD degrees from the University of Mississippi and the University of Oklahoma, respectively, both in Computer Science. His research interests include object-oriented software engineering, programming language theory and distributed systems. He is a member of the Software Engineering Association of Taiwan (SEAT).

I-Lang Wu received his MS degree from the National Taipei University of Technology, Taiwan, in Computer Science. He is an independent software consultant focusing on the banking domain. Prior to his consultant profession, he served as an associate architect at IBM Taiwan. His research interests include software architecture and legacy system maintenance. He is a member of the Software Engineering Association of Taiwan (SEAT).