Exception Handling: An Architecture Model and Utility Support

Y. C. Cheng and Chien-Tsun Chen
Department of Computer Science and
Information Engineering
National Taipei University of Technology
Taipei, Taiwan 106
{yccheng, s1669021}@ntut.edu.tw

Jung-Sing Jwo
Department of Computer Science and
Information Engineering
Tunghai University, Taichung, Taiwan
jwo@mail.thu.edu.tw

Abstract

Exception handling design is an important but difficult subject in software development. In Java software development, the use of checked exceptions exacerbates the difficulty. In this paper, through the use of an architectural model, we show that an application can benefit from a separation of exceptions in terms of recoverability beyond distinguishing checked and unchecked exceptions. The architectural model helps evaluating and balancing conflicting quality requirements such as modifiability, readability, and fault tolerance. Facilitated by object-oriented utility libraries, the architecture model guides the design from early stage of the development; an example is given to illustrate its use.

1. Introduction

Exception handling has consistently attracted attentions from the programming community. While certain design principles are generally accepted as good practices [2, 9, 10, 11], a consensus on what constitute the ultimate set of principles is non-existent. In Java, the most noticeable split is the use of checked and unchecked exceptions. Some argued that checked exceptions clutter the object interfaces and induce complicated catch blocks and should be wrapped as unchecked exceptions, which are caught only at the top-level architectural components [1, 2, 13]. Others argue that checked exceptions are a required language feature to effectively fulfill fault tolerance in the application [1, 2, 12].

A decision on the use of exceptions is application-dependent [12] and is often pulled in different directions by a number of quality requirements including modifiability, readability, and fault tolerance. It is probably safe to say that a real life design will usually resolve by achieving each quality requirement to some extent; moreover, the resolution itself can change over time. Thus, what is needed is some architectural and design supports that will guide architects and designers in exception handling design and evolution through the life cycle of the software.

In the paper, we propose an architecture-level model of exceptions and object-oriented utility supports for exception handling. The architectural model consists of the exception archetypes, their intended meanings, and relationships among them. By adding and deleting components and connectors, architects and designers can have a high-level view of the impact of exception handling policies on the quality requirements. In this way, the architectural model helps balancing quality requirements of the software under development. The architectural model also contributes to the evolution of the object-oriented designs of the exception hierarchy and design for exception handling.

The rest of this paper is organized as follows. Section 2 reviews the Java exception handling mechanism and proposes the concept of exception architectures. In Section 3, the proposed architecture model for exceptions is presented. The object-oriented supporting utility for exceptions is discussed in Section 4, and an example that demonstrates its usage is illustrated in Section 5. Section 6 gives a conclusion and indicates possible future research directions.

2. Exception handling in Java

The Java language has three categories of exceptions, namely, Exception, RuntimeException, and Error. The first category is called checked exception and the latter two are unchecked exceptions. The Java compiler treats checked exceptions differently from the unchecked ones. When a checked exception is thrown inside a method, it has to be either caught or declared in the method’s signature. This is the so-called “handle-or-declare” rule [10] and is enforced by the Java compiler. For unchecked exceptions the rule does not apply.

The semantics of these three exception categories are defined in the Java Language Specification [6]. In a nutshell, exceptions, such as IOException and SQLException that are potentially recoverable belong to Exception. RuntimeException, for example NullPointerException, usually indicates a bug in programs that could cause by any statement. In general, such an exception should be fixed rather than catch and recover in code. Error, such as StackOverflowError and OutOfMemoryError represents disasters that occur within the underlying JVM and operating system. Although these definitions and many exception handling best practices have been proposed, in practice there are still some common problems that occur over and over, including empty catch blocks, blanket catches, blanket declarations, duplicated and complex handling code, incomplete error logs, and so on [9, 10]. We argue that without an architectural perspective in mind these best practices and guidelines could be followed inconsistently. Thus, the robustness and exceptional behaviors of the developed system may be difficult to anticipate.

In the context of exception handling, we are concerned with two architectural perspectives: application architectures and exception architectures. Application architectures provide a global context that will influence an application’s exception handling policies. For example, in a layered architecture a lower layer component is trying to create a server socket to accept connections at port 80. Suppose that the port is already in use by another application and a BindException is raised. Although the component is the first to catch the exception, it does not have enough contexts to resolve it and a checked exception is thrown. Treatment of the exception takes place in the upper application layer, which has a more complete context. If the treatment failed, an exception may be fired up to the presentation layer and an end-users readable error dialog is pop-up. Without the application architecture, only local context is available and the exception handling policies tend to be arbitrary.

The exception architectures provide a conceptual model to systematic treatment of exceptions. Basically, we introduce three architectural models, fault reporting, partial fault tolerance, and fault tolerance. We present these models in section 3. These models act as containers that can accommodate commonly used exception handling policies in Java. Different types of applications can use different modes according to their characteristics. For example, a query-based web application can use the fault reporting model for simplicity; an online game application may use fault tolerance model at server side and fault reporting model at client side.

3. The architecture model for exceptions

In software development, an architectural model partitions the functionality into modules and allows quality requirements to be specified, reasoned, and evaluated even before any program code is written. Since exception handling involves functionality upon errors and increases code complexity in a program, an architectural model for exception should be produced alongside the software architecture of the application at the onset. This section presents the static and dynamic architectural models for exception handling, and discusses the related design issues in various ways of realizing the architecture.

3.1 The Static model

In Figure 1, the architectural model distinguishes mechanism exceptions and application exceptions: the former are exceptions encountered in the implementation of an application component making calls to the Java or third-party API’s; the latter are those raised by the components of the application. Mechanism exceptions, which are either checked or unchecked, are converted to application exceptions when it is caught. Application exceptions are further classified based on recoverability. An irrecoverable exception, which is derived from RuntimeException, is propagated without intervention until the top of the call chain is reached, where it is caught and handled. A recoverable exception, which is derived from Exception, is handled by some handler with three possible outcomes: (1) the exceptional condition is successfully resolved; (2) the exception is determined to be not fixable and a wrapping irrecoverable exception is raised; and (3) the handler neither fixes nor raises an irrecoverable exception, but reflects the recoverable exception. Useful handling policies can be applied, for example, in the resolution option, to retry or to return a default object that is acceptable though not the most exact.
Essential characteristics are determined by adding or deleting components and relationships in the architectural model. In the fault tolerance model of Figure 1, the relationships from mechanism exceptions to application exceptions have the greatest impact on the complexity of the try blocks, since mechanism exceptions must be caught and reflected as application exceptions. In the case of unchecked exceptions, the overhead can be forbidding since any method can throw unchecked exceptions.

For simplicity of try blocks and scalability of exception handling, the unchecked exceptions are not caught until at the top of the call chain. On the other hand, the checked exceptions are caught and converted into irrecoverable exceptions, as in Figure 2, to achieve fault reporting. Note that the model reflects design practice to wrap checked exceptions as unchecked exceptions, which are caught only at the top-level architectural components.

Should situations arise to demand some degree of fault tolerance, it is necessary to have the recoverable exception in the architectural model. To avoid pervasive try-blocks in catching unchecked exceptions, only checked exceptions are caught and converted into recoverable or irrecoverable exceptions. This achieves partial fault tolerance. The result is shown in Figure 3.

3.2 Behavioral model

The behavioral aspect of the exception model is shown in Figure 4, which reflects the fully fault-tolerant model of Figure 1 but can be simplified to reflect the failure-reporting model and the partially fault-tolerant model in Figures 2 and 3, respectively. The scenario illustrates how two domain objects interact during exception handling. Domain object A invokes a method of domain object B (1), which calls the Java or a third party API (2). A mechanism exception, a checked exception in this case, is raised (3) and caught by B and handled in a catch block (4). The catch block has three options: to throw a recoverable exception (5a), to throw an irrecoverable exception (5b), or to attempt a retry (5c). If a retry is attempted, the catch block further specifies either to throw a recoverable exception (5c.a) or an irrecoverable exception (5c.b) if the retry fails. If the retry is
successful (5c.c), normal logic resumes after completion of the try block (6c).

The architectural models can be used to create Java applications with clear exception handling policies. In the next section, we show how to further take advantage of the model with a utility that takes care of the conversion and retry logic and user level resolutions.

3.3 Design issues for the architecture

Application frameworks represent the highest level of reuse and enforcement of design guidelines imposed by the architecture. However, we shall argue here that the framework is not the best choice for exception handling.

Moreover, the framework usually requires some kinds of configuration mechanisms to map an exception to a handler. Such mapping mechanisms tend to require programmers to use a single generic catch (blanket catch) in code. This is equivalent to deactivate the compile time checking for checked exceptions, which distinguishes Java from other language such as C++ and C#. [7] and [8] demonstrate frameworks that roughly follows the above design principle.

In our design, we focus on embedding exception architecture models into an object-oriented class library to aid programmers in writing exception handling code.

4. Supporting utility

Two levels of utilities are supported in two packages: core for general exception handling activities and resolution for exception resolutions at the user interface level.

4.1 The core package

Figure 5 shows the conceptual diagram for the core package. Three basic exceptions are defined in this package as shown in Figure 6. In the following, primary components are discussed.

![Figure 4. Behavioral model](image)

![Figure 5. Conceptual diagram of core package](image)

![Figure 6. Basic exceptions of core package](image)
**EHCORE**: EHCORE is a factory and static class. It is used to create instances of four predefined services, including IRepeatExecutor, IExceptionChecker, ILogger, and IUncaughtExceptionHandler. These services are defined as public Java interfaces, which programmers use to access the services. However, the implementation is private to programmers and can be changed form release to release.

**IRepeatExecutor**: One common approach to handle exceptions is performing the original action again, i.e., the so-called retry model. However, since Java uses the termination model, programmers have to write their own code to simulate the retry model. IRepeatExecutor encapsulates the logic for time redundancy (performed at a regular time interval), which is a general approach to software fault tolerance. Three time redundancy models are supported: (1) retry, executing the original action with the original parameters; (2) default, executing the original action but using default values to replace the original parameters; (3) alternative, invoking another method instead of the original one.

**ILogger**: Exception logging is another common practice when handling exceptions. It can provide useful information for testing and debugging, especially for irrecoverable exceptions. ILogger takes care of over-logging, a common problem in exception logging, where an exception is logged more than once in a log file [9]. Note that an over-logged exception is tagged in the log file.

**IExceptionChecker**: To hide methods’ implementation details and to allow exception evolution, a method should not declare exception types that are too specific. For example, most classes in the Java IO package declare to throw an IOException even though they throw subtypes of IOException. This is usually fine for component designers, but can be confusing for component users if they want to provide refined handling behavior for a specific exception. IExceptionChecker is designed to check this condition. For example, if a catch clause catches an IOException, but at runtime an instance of BindException (which is a subclass of IOException) is passed, IExceptionChecker can report this situation.

**IUncaughtExceptionHandler**: IUncaughtExceptionHandler provides a default handler for uncaught exceptions. It logs the uncaught exceptions and may show an error dialog.

**RepeatExecutorFailureRecoverableException**: When using IRepeatExecutor to perform time redundancy activities, the executed actions can fail as well. The RepeatExecutorFailureRecoverableException is a checked exception to indicate that a method fails to provide the intended fault-tolerance behavior but its caller may try to recover from this failure if possible. This exception is generally thrown by invoking IRepeatExecutor.run().

**RepeatExecutorFailureIrrecoverableRTE**ception: Though similar to RepeatExecutorFailureRecoverableException, the exception is an unchecked exception and indicates a failure recovery should not be attempted any more. It is generally thrown by invoking IRepeatExecutor.runIrri().

**ProgrammingRE**ception: This exception represents a logical error in code, i.e., a bug to be fixed. Its use is similar to the assert statement in Java or the check statement in Eiffel except that its execution cannot be turned off at runtime. Programmers can put a throw new ProgrammingRTEception (“reason”) statement at an unreachable execution path to indicate a bug. A common use of this statement is to prevent the empty catch (catch and do nothing) symptom. Experiences show that programmers tend to catch checked exceptions blindly so as to pass the compilation and may forget to write handling code later. Putting such a statement in an empty catch block reminds programmers that their duties are not finished.

### 4.2 The resolution package

Generally speaking, once exceptions are propagated to the user interface layer and error messages are displayed to the users, the alternative is limited. Aborting, ignoring, and retrying are typical solutions for non-critical exceptions. For critical exceptions, the application may have to be terminated and restarted. Recently, more and more applications are designed to provide “active help” so that users can easily repair the exception and resume their work more quickly. To enhance the application's intelligence, multiple resolutions may be provided for a specific exception type. Users then decide which action to take to resolve this exception. The resolution package, which is inspired by the Marker and Resolution mechanism in Eclipse [4, 5], provides such supports.

Figure 7 shows the conceptual diagram for the resolution package. Four components are discussed. **EHResolutionFactory**: When an IExceptionResolutionGenerator is created to generate resolutions for a specific exception, it has to register itself to EHResolutionFactory. Later, when EHResolutionFactory is required to display a ResolutionDialog for a particular exception, it asks for each registered IExceptionResolutionGenerator instance to provide suitable resolutions.
IExceptionResolutionGenerator: It binds an exception to potential resolutions. Two methods are defined in this interface: boolean hasResolutions(Exception) and IExceptionResolution[] getResolutions(Exception).

IExceptionResolution: The resolution logic for an exception is defined in classes that implement this interface. It defines two callback methods: getDescription for getting description message for that resolution and run(Exception e) for executing the treatment activities. Note that the resolution classes usually have to access the corresponding domain objects to perform necessary treatment. Thus, they are not design for reuse.

ResolutionDialog: The responsibility of ResolutionDialog is to provide the user interface for users to perform a resolution. Although multiple resolutions for a giving exception are possible, users can only select one to execute.

Since a game server hosts multiple sessions concurrently, an exception should not cause the whole application to crash. Because an Acceptor is located at the bottom layer and is designed for reuse, the treatment is out of its context. Thus, the best exception handling policy for an Acceptor is propagation. In this example, the Acceptor merely re-throws the original exception rather than wrap it into another refined type. Figure 9 shows the code snippet of the Acceptor.

The exception is propagated to the application layer and caught by the GameServer, which tries to recover from this exception. Three possible solutions may be used, including retry, use a default value, and take an alternative (N-version programming). Figure 10 shows the code snippet of the GameServer’s createAcceptor method. It uses the default value policy to handle the exception if invoking Acceptor’s createSocket method raises an exception. Should the treatment fail, it throws an application-level exception CannotCreateAcceptorException and leaves it to AppWin to solve the problem.

Once the AppWin catches the CannotCreateAcceptorException, it may directly report this exception by providing a readable description to users. Figure 11 shows the code snippet that demonstrates the use of ResolutionDialog to provide active fix through interaction with users. Figure 12 is the screenshot of executing ResolutionDialog.show().

public class Acceptor {
    // Implementation...
}
ServerSocket fSocket;

public void createSocket(Integer port) throws IOException{
    fSocket = new ServerSocket(port);
}

Figure 9. Code snippet of Acceptor

public void createAceptor(Integer port) throws CannotCreateAceptorException{
    Acceptor acceptor = new Acceptor();
    IRepeatExecutor repair = EHCore.getRepeatExecutor();
    repair.setAttempt(2);
    repair.setMode(IRepeatExecutor.Mode.defaultValue);
    repair.addDefaultArgs(port + 1);
    repair.addDefaultArgs(port + 2);
    try { repair.run(acceptor, "createSocket", port); } catch (RepeatExecutorFailRecoverableExcepton e){ throw new CannotCreateAceptorException(e); }

} Figure 10. Code snippet of GameServer

// setup resolution generator
EHResolutionFactory.addResolutionGenerator(new CannotCreateAceptorResolutionGenerator()); ...

try{ // call GameServer
} catch(CannotCreateAceptorException e){
    // display ResolutionDialog
    EHResolutionFactory.getResolutionDialog(e).show();
}

Figure 11. Code snippet of AppWin

6. Conclusion and future work

Although exception handling is critical for developing robust applications, it is generally neglected by many software developers. We argued that effective exception handling should consider application and exception architectures, and presented three evolvable exception architecture models. These models can guide developers to consistently use check and unchecked exceptions in Java. We further presented utilities that support common exception handling activities, including time redundancy fault tolerance, logging, generic catch checking, and default exception handing. A user interface layer package that provides sophisticated exception recovery mechanism was proposed as well.

We expect that the exception architectures and the exception handling utilities can aid software participants in systematically developing robust applications. Future work includes developing an Eclipse plug-in to help programmers in writing exception handling code and porting the utility to other languages, such as C++ and C#.

7. References


