

Error Handling as an Aspect

Fernando Castor Filho¹, Alessandro Garcia², Cecília Mary F. Rubira³

¹ Department of Computer Science, University of São Paulo
Rua do Matão, 1010. 05508-090, São Paulo, SP, Brazil

² Computing Department, Lancaster University
South Drive, InfoLab 21, LA1 4WA, Lancaster, UK

³ Institute of Computing, State University of Campinas
P.O. Box 6176, 13084-971, Campinas, SP, Brazil

fernando@ime.usp.br, garciaa@comp.lancs.ac.uk, cmrubira@ic.unicamp.br

ABSTRACT

One of the fundamental motivations for employing exception handling in the development of robust applications is to lexically separate error handling code from the normal code so that they can be independently modified. However, experience has shown that exception handling mechanisms of mainstream programming languages fail to achieve this goal. In most systems, exception handling code is intertwined with the normal code, hindering maintenance and reuse. Moreover, because of limitations in the exception handling mechanisms of most mainstream programming languages, error handling code is often duplicated across several different places within a system. In this paper we present a pattern, **Error Handling Aspect**, which leverages aspect-oriented programming in order to enhance the separation between error handling code and normal code. The basic idea of the pattern is to use advice to implement exception handlers and pointcuts to associate advice to different parts of the normal code in order to improve the maintainability of the normal code and promote reuse of error handling code.

Categories and Subject Descriptors

D.2.3 [Software Engineering]: Coding Tools and Techniques

General Terms

Languages, Design

Keywords

exception handling, aspect-oriented programming

1. INTRODUCTION

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Workshop BPAOSD'07, March, 12-13, 2007 Vancouver, BC, Canada
Copyright 2007 ACM 1-59593-662-2/07/03 ...\$5.00.

Exception handling [7] mechanisms were conceived as a means to structure programs that have to cope with erroneous situations. These mechanisms make it possible for developers to extend the interface of an operation with additional exit points that are specific to error recovery. Moreover, they define new constructs for raising exceptions and associating exception handlers with selected parts of a program. Ideally, an exception handling mechanism should enhance attributes such as reliability, maintainability, reusability, and understandability, by making it possible to write programs where: (i) error handling code and normal code are lexically separate and can be maintained independently [17]; (ii) the impact of the code responsible for error handling in the overall system complexity is minimized [18]; and (iii) an initial version that does little recovery can be evolved to one which uses sophisticated recovery techniques without a change in the structure of the system [17].

1.1 Problem

Even though separation of concerns is the overarching goal of exception handling, the kind of separation promoted by the exception handling mechanisms of most mainstream object-oriented programming languages brings only limited advantages [3, 8, 15]. In languages such as Java, Ada, C++, and C#, it is not possible to “plug” and “unplug” exception handlers. In these languages, the normal code and error handling code are entwined within fine-grained units (code blocks), making it hard to maintain the former independently from the latter. Also, this hardwiring of the exception handling code to the normal code hinders reuse of normal code across different applications, as these applications often have different requirements pertaining error handling.

Another problem is that the exception handling mechanisms of the aforementioned languages only support the definition of handlers that are local to specific parts of a program. Reuse of error handling strategies within an application is possible only to a certain degree, by extracting error handling measures to new methods. However, in most mainstream programming languages, the code that catches exceptions and initiates an exception handling measure has to be scattered throughout the application. As a consequence, most systems have a considerable amount of duplicated exception handling code. For example, consider the following Java code snippet, extracted from an Eclipse plugin:

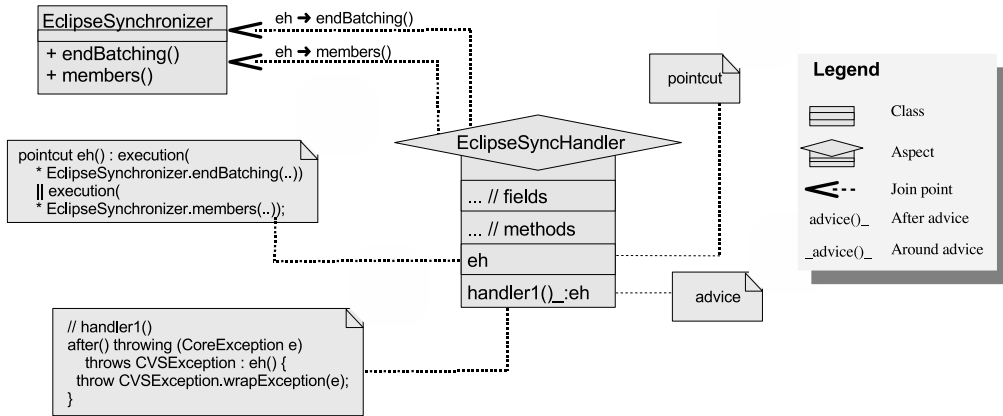


Figure 1: An example where the use of Error Handling Aspect avoids duplication of exception handling code.

```

public class EclipseSynchronizer implements IFlushOperation {
    public void endBatching(...) throws CVSException {
        try {...} catch (TeamException e) {
            throw CVSException.wrapException(e);
        }
    }
    ...
    public IResource[] members(...) throws CVSException {
        ...
        try {...} catch (CoreException e) {
            throw CVSException.wrapException(e);
        }
    }
    ...
}
  
```

In the example, two different methods within the same class, `endBatching()` and `members()`, implement identical exception handling strategies. `TeamException` is a subtype of `CoreException`. In Java, it is not possible to implement a single handler and associate it to both methods, to avoid code duplication.

1.2 Solution

The Error Handling Aspect design pattern leverages features of aspect-oriented languages to promote an explicit separation between exception handling code and normal code. The overall idea of the pattern is to use advice to implement exception handlers and associate these “aspectized” handlers to different parts of a program by means of the composition mechanisms provided by aspect-oriented languages. Figure 1 shows how the pattern solves this problem. It uses a slightly modified UML notation derived from the notation proposed by Chavez [5].

In Figure 1, aspect `EclipseSyncHandler` defines a pointcut named `eh` that associates advice `handler1` to methods `members()` and `endBatching()`. This advice implements the exception handlers that would otherwise be scattered throughout the application code and is therefore called a *handler advice*. The name of each advice in the diagram is followed by the name of a pointcut to which it is bound. The code snippets in the comments correspond to possible implementations written in AspectJ [14]-like languages. This approach separates the error handling code from the normal code and localizes it within a single program unit, namely, an error handling aspect implementing the various handler advice.

2. STRUCTURE

Figure 2 presents the overall structure of the pattern. Aspects `EHAspect1`, `EHAspect2`, and `GenericEHAspect`, include each a single handler advice, `handler1()`, `handler2()`, and `genericHandler()`, respectively. The join points of interest, in this case, are methods that throw exceptions. In the rest of this paper, we call “exception-throwing statement” a statement that potentially throws an exception. Exception-throwing statements appear within “context methods”. We use this term because, usually, these methods define exception handling contexts. In the figure, classes `NormalClass1` and `NormalClass2` define one context method each, `contextMethod1()` and `contextMethod2()`, respectively. We refer to a set of exception-throwing statements within the same context method as “exception-throwing code”. Besides pointcuts and advice, error handling aspects can also include methods and fields that are specific to exception handling. A field in this case can be, for example, a hash table that stores temporary values that the handler advice use.

Normal classes. Classes `NormalClass1` and `NormalClass2` implement the normal code of an application. Each has one or more context methods, including one or more exception-throwing statements apiece.

Concrete error handling aspects. In Figure 2, there are two concrete error handling aspects, `EHAspect1` and `EHAspect2`. Each one includes one or more handler advice. The handler advice implement exception handling code that is executed when exceptions are raised within the context methods. A handler advice may be bound to several distinct context methods, in order to avoid duplicating exception handling code.

Abstract error handling aspects. If a handler advice is common to two or more error handling aspects, it is useful to move it to an abstract aspect and make the latter a super-aspect of the other error handling aspects. By binding the advice to an abstract pointcut and making it concrete in the sub-aspects, duplication of handler advice is avoided. In Figure 2, aspect `GenericEHAspect` is an example of abstract error handling aspect.

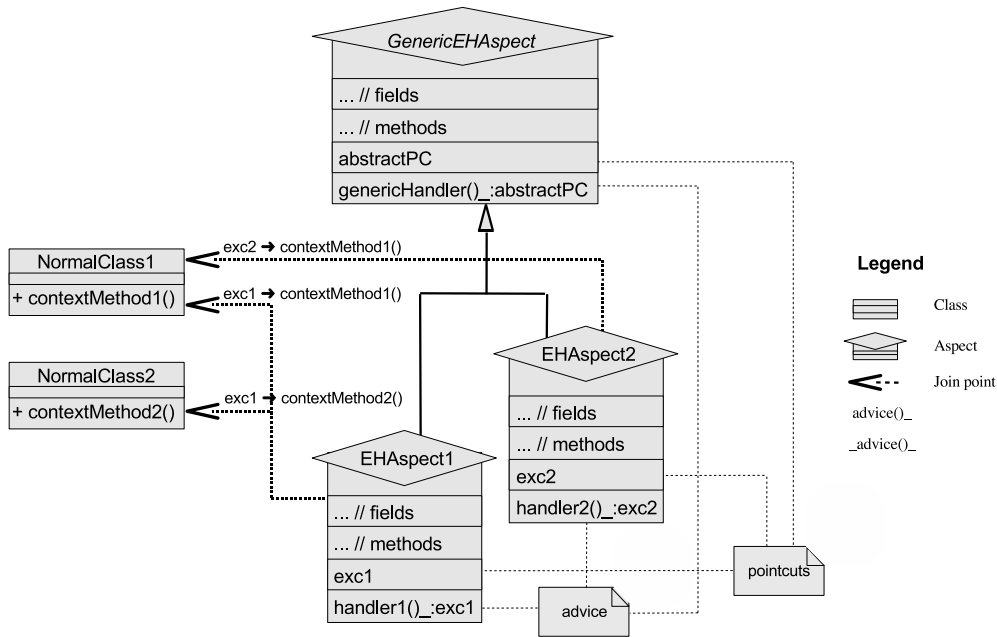


Figure 2: General structure of the pattern.

One of the driving forces behind aspect-oriented techniques is the idea that the base code can be oblivious to aspects [10]. However, in our experience [3], the best results are achieved when the base code is designed with aspects in mind from the start. In the specific case of error handling, this means that designers of the base code should try to: (i) expose join points of interest that aspect-oriented languages can easily capture; and (ii) make it easy for the error handling aspects to access the contextual information they need. Otherwise, the complexity of error handling aspects will grow and the base code will have to be modified in order to use the pattern.

3. DYNAMICS

The following scenarios illustrate how the various components of the Error Handling Aspect pattern interact at runtime.

Scenario 1. Figure 3 depicts the scenario where a client invokes a method on a certain object, an exception is raised while the method is being executed, and an error handling aspect successfully handles the exception.

1. A client object invokes method `contextMethod()` on an instance of `NormalClass`. As implied by the name of the method, exceptions might potentially be raised within it.
2. While the method is being executed, exception `E` is raised.
3. Control is transferred to `EHAAspect`, an error handling aspect which attempts to handle `E`.
4. The handler ends its execution normally, without raising any exceptions.

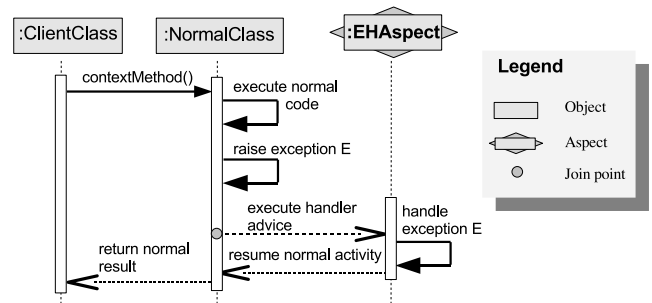


Figure 3: A scenario where a handler advice successfully handles an exception.

5. Control returns to the normal code, which resumes execution. Depending on the join point to which the handler is bound, this means that either `contextMethod()` goes on executing or that it immediately returns some normal (non-exception) response to the client object.

Scenario 2. This scenario shows the dynamics of error handling aspects that simulate two nested `try-catch` blocks. Advice `innerHandler()` and `outerHandler()` in Figure 4, defined by aspects `InnerEHAAspect` and `OuterEHAAspect`, respectively, are associated to the same join point. More specifically, the scenario illustrates the case where the inner handler advice, after failing to handle an exception thrown within a context method, throws an exception that is caught by the outer handler advice. The latter then signals an exception which is received by the client object.

1. A client object invokes `contextMethod()` on an instance of `NormalClass`.

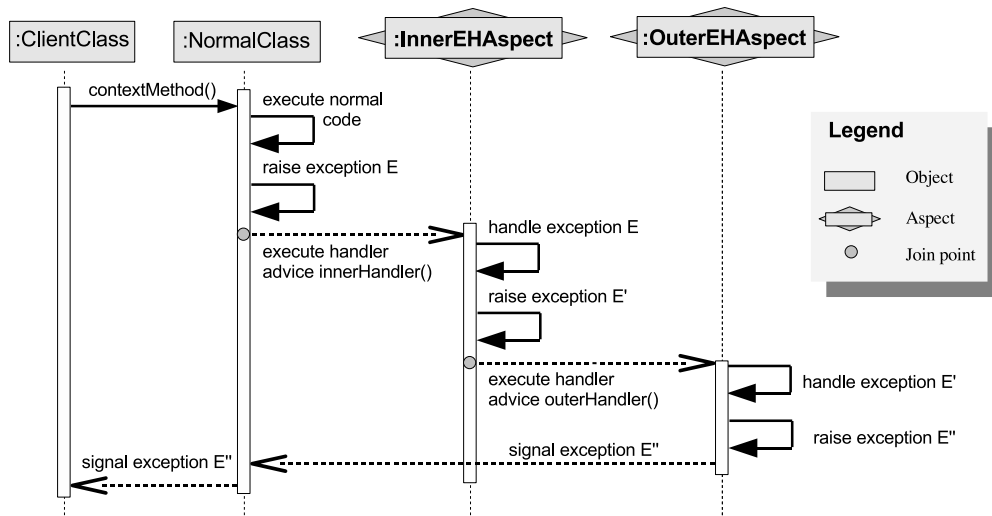


Figure 4: A scenario involving nesting of handler advice.

2. While the method is being executed, exception `E` is raised.
3. Control is transferred to the `innerHandler()` handler advice, which attempts to handle `E`.
4. Handler advice `innerHandler()` raises exception `E'`.
5. Control is transferred to the `outerHandler()` handler advice, which attempts to handle `E'`.
6. Handler advice `outerHandler()` raises exception `E''`.
7. Exception `E''` is signaled to the `NormalClass` object, re-signaled by the latter, and finally received by the client object.

4. CONSEQUENCES

The Error Handling Aspect pattern has the following *benefits*:

- *Localization of error handling code.* An important benefit of Error Handling Aspect is that it keeps all the exception handling code localized within program units whose sole purpose is to implement the exception handling concern. In other words, it reduces the scattering of exception handling code. This localization eases system maintenance, as developers do not have to search through a whole program in order to change a certain exception handler. It also improves understandability, since it is possible to get an intuitive understanding of how error handling works in a given system just by looking at the error handling aspects.
- *Reduction of duplicated error handling code.* It is easy to encapsulate an exception handler that would otherwise appear in several parts of a system in a single handler advice. These parts of the system then become the join points of interest to which the handler advice will be associated. This reduction of duplicated error handling code does not necessarily mean that the pattern will reduce the overall number of lines of code

pertaining error handling, though. Since error handling is a very context-specific concern, it is often the case that similar exception handlers cannot be combined to form a single handler advice. Moreover, the use of AOP incurs in an overhead due to the implementation of pointcuts and advice. In our experience, the overall size of programs tends to grow with the use Error Handling Aspect [3].

- *Arbitrary exception handling contexts.* The fundamental precept of the Error Handling Aspect pattern is that advice implement exception handlers and are associated to exception throwing code through pointcuts. Because of this, the only limitation to the types of exception handling contexts that can be defined is the join point model of the employed aspect-oriented language.
- *Pluggability.* An error handling aspect can be easily replaced by another error handling aspect implementing different error handling strategies. This feature makes it easy to reuse the normal code of an application or part of it across different systems. The capability of reusing the normal code separately from the error handling code is desirable in cases where different systems require specific error handling strategies.
- *Textual separation.* Arguably, the textual separation promoted by Error Handling Aspect makes it easier to understand how a system works. The rationale is that developers have to grasp smaller and more clearly-defined conceptual units that implement specific concerns. In other words, the pattern reduces the tangling of exception handling code.

Additionally, Error Handling Aspect has the following *liabilities*:

- *Difficulty in visualizing the effects of aspects.* In spite of the advantages of textual separation, it makes it difficult for a developer examining the base code of an application to have a complete understanding about

system behavior. Getting a complete picture requires an understanding about base code, aspects, and their often non-obvious interactions. It is often argued that tool support can help developers in overcoming this problem [15], but current tools are still not mature enough.

- *Aspect invasiveness in some scenarios.* In some recurring situations, current aspect-oriented languages cannot simulate the exception handling mechanisms of existing programming languages. The design of the base code must take this into account and avoid these situations. Otherwise, **Error Handling Aspect** cannot be applied. When extracting error handling code from an object-oriented implementation in order to use **Error Handling Aspect**, this means that sometimes the system has to be refactored a priori, before the exception handling can be “aspectized”. In other words, it is often the case that the base code cannot be oblivious to aspects [20] and must be prepared a priori.
- *Limited integration with checked exceptions.* In languages that use checked exceptions, such as Java, a method is required to either handle all the checked exceptions it encounters or explicitly declare those it does not in its interface, otherwise the compiler will complain. Hence, some aspect-oriented languages, such as CaesarJ [16] and HyperJ [21], can only be used in situations where the “aspectization” of error handling results in programs whose base (non-aspect) code does not violate the language rules for checked exceptions. AspectJ provides a workaround for this problem called *exception softening*. This language feature makes it possible to suppress the checks conducted by the Java compiler in certain join points. Therefore, the use of **Error Handling Aspect** in AspectJ requires that all exceptions caught by handler advice become unchecked.
- *Increase in program size.* In the early days of AOP, it was often claimed that its use for structuring exception handling code would result in a reduction in application size [15]. However, more recent studies [2, 3] have shown that this is only true if error handling code is uniform and context-independent. If exception handling code in an application is non-uniform or strongly context-dependent, reuse of handler code becomes low and the application size can grow due to the implementation overhead of AOP. More specifically, the number of operations (methods and advice) and components (aspects and classes) will almost always grow due to the use of **Error Handling Aspect**.

5. KNOWN USES

Lippert and Lopes [15] were the first to report to a broader audience on the use of AOP to modularize error handling. They applied the pattern to an object-oriented framework called JWAM using an old version of AspectJ. Colyer and Clements [6] employed **Error Handling Aspect** to capture data about component failures in a commercial middleware infrastructure. Due to application requirements, they could encapsulate all the error handling strategies of the application within a single abstract aspect, maximizing reuse of handler code.

Soares and colleagues [19] used **Error Handling Aspect** to structure part of the exception handling code in a web-based healthcare information system named Health Watcher. This work distinguishes itself from the ones mentioned above because the authors targeted specifically the exceptions introduced in the system by distribution and persistence concerns. In Health Watcher, these two concerns were implemented as aspects.

Castor Filho et al [3] used this pattern to structure error handling in four different systems: (i) a web-based traveller information system; (ii) Java Pet Store¹, a well-known demo for the Java Platform, Enterprise Edition; (iii) the CVS Core Plugin, part of the basic distribution of the Eclipse² platform; and (iv) Health Watcher [19]). The first three applications were originally object-oriented, whereas the fourth included some concerns that were implemented a priori as aspects. They also empirically analyzed the impact of the pattern in these four systems based on a set of metrics for quality attributes such as coupling, cohesion, and conciseness.

6. RELATED PATTERNS

Error Handling Aspect presents some improvements over the **Handler** pattern, proposed by Garcia and Rubira [12]. **Handler** leverages a meta-object protocol in order to promote a complete textual separation between normal code and error handling code. One of the differences between **Handler** and **Error Handling Aspect** is that, in the latter, the use of aspects makes it possible to define arbitrary, both fine- and coarse-grained, exception handling contexts. The only limitation to what can be selected as an exception handling context is the join point model of the employed aspect-oriented language. Moreover, the quantification capabilities of aspect-oriented languages arguably make it easier to localize error handling code within the aspects. Also, using the **Error Handling Aspect**, the pointcut descriptions explicitly point out the locations where the classes and error handling aspects interact. In a reflective solution, these interactions are intertwined/hardcoded in the method body of meta-objects.

The **Exception Introduction** pattern [14] leverages aspect-oriented programming to make new exceptions introduced by aspect-oriented implementations of crosscutting concerns transparent to the base code of an application. The pattern targets languages such as Java, which use checked exceptions, and makes the introduced exceptions temporarily unchecked so that they can be handled where it is more appropriate. **Exception Introduction** uses **Error Handling Aspect** to implement the exception handlers for the introduced exceptions.

Many authors [9, 14, 15] propose the use of AOP for separating runtime assertion-checking code from the normal code. This pattern can be used in combination with **Error Handling Aspect** so that both error detection and error handling code become localized within well-defined program units. This combined solution results in normal code that is not cluttered by error detection and handling concerns.

An extended version of this paper [4], including a detailed discussion of implementation issues, was submitted to the *6th SugarloafPLoP conference*.

¹<http://java.sun.com/developer/releases/petstore/>

²<http://www.eclipse.org>

7. ACKNOWLEDGEMENTS

This work was conducted while Fernando was with the Institute of Computing, State University of Campinas, supported by FAPESP/Brazil, grant 02/13996-2. Alessandro is supported by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008. Alessandro is also supported by the TAO project, funded by Lancaster University Research Committee. Cecilia is partially supported by CNPq/Brazil, grant 351592/97-0, and by FAPESP/Brazil, grant 2004/10663-8.

8. REFERENCES

- [1] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Springer-Verlag, 2nd edition, 1990.
- [2] Thiago Tonelli Bartolomei. On modularity assessment of aspect-oriented software. Master's thesis, Kiel University of Applied Sciences, Kiel, Germany, October 2006.
- [3] F. Castor Filho, N. Cacho, E. M. Figueiredo, R. M. Ferreira, A. Garcia, and C. M. F. Rubira. Exceptions and aspects: The devil is in the details. In *Proceedings of the 14th SIGSOFT FSE*, pages 152–162, Portland, USA, November 2006.
- [4] F. Castor Filho, A. Garcia, and C. M. F. Rubira. The Error Handling Aspect pattern, 2007. Submitted to the 6th Latin-American Conference on Pattern Languages of Programs (SugarloafPLoP'06).
- [5] C. Chavez. *A Model-Driven Approach for Aspect-Oriented Design*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brazil, April 2004.
- [6] A. Colyer and A. Clement. Large-scale AOSD for middleware. In *Proceedings of AOSD'04*, pages 56–65, March 2004.
- [7] F. Cristian. Exception handling. In *Dependability of Resilient Computers*. BSP Professional Books, 1989.
- [8] Q. Cui and J. Gannon. Data-oriented exception handling. *IEEE Transactions on Software Engineering*, 18(5):393–401, May 1992.
- [9] F. Diotalevi. Contract enforcement with aop, July 2004. IBM DeveloperWorks - <http://www-128.ibm.com/developerworks/library/j-ceaop/>.
- [10] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proceedings of the Workshop on Advanced Separation of Concerns, OOPSLA'2000*, October 2000.
- [11] A. Garcia, C. Rubira, A. Romanovsky, and J. Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2):197–222, 2001.
- [12] A. Garcia and C. M. F. Rubira. An architectural-based reflective approach to incorporating exception handling into dependable software. In A. Romanovsky et al., editors, *Advances in Exception Handling Techniques*, LNCS 2022. Springer-Verlag, 2000.
- [13] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, December 1975.
- [14] R. Laddad. *AspectJ in Action*. Manning, 2003.
- [15] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd ICSE*, pages 418–427, June 2000.
- [16] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Proceedings of the 2nd AOSD*, pages 90–99, March 2003.
- [17] D. L. Parnas and H. Würges. Response to undesired events in software systems. In *Proceedings of the 2nd ICSE*, pages 437–446, San Francisco, USA, October 1976.
- [18] B. Randell and J. Xu. The evolution of the recovery block concept. In *Software Fault Tolerance*, chapter 1, pages 1–21. John Wiley Sons Ltd., 1995.
- [19] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the 17th OOPSLA*, pages 174–190, 2002.
- [20] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In *Proceedings of the 10th ESEC/13th SIGSOFT FSE*, pages 166–175, Lisbon, Portugal, 2005.
- [21] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st ICSE*, pages 107–119, May 1999.

9. APPENDIX - EXCEPTION HANDLING

Exception handling [7, 13] is a mechanism for structuring error recovery in software systems so that errors can be more easily detected, signaled, and handled. It is implemented by many mainstream programming languages, such as Java, Ada, C++, and C#. These languages allow the definition of exceptions and their corresponding handlers. The set of exceptions and exception handlers in a system define its abnormal or exceptional activity.

When an error is detected, an exception is generated, or *raised*. If the same exception may be raised in different parts of a program, different handlers may be executed, depending on the place where the exception was raised. The choice of the handler that is executed depends on the exception handling context where the exception was raised. An exception handling context is a region of a program where the same exceptions are handled in the same manner. Each context has an associated set of handlers that are executed when the corresponding exceptions are raised. Typical examples of exception handling contexts in object-oriented languages are blocks, methods, classes, and exceptions [11].

The *idealized fault-tolerant component* (IFTC) [1] defines a conceptual framework for structuring exception handling in software systems. An IFTC is a component (in a broader sense – an object, a software component, a whole system, etc.) where the parts responsible for the normal and abnormal activities are separated and well-defined, within its internal structure. The goal of the IFTC approach is to provide means to structure systems so that the impact of error recovery mechanisms in the overall system complexity is minimized. One of the main motivations of **Error Handling Aspect** is to promote the construction of systems where all the system components are IFTCs.