An Exception Handling Software Architecture for Developing Robust Software

Alessandro F. Garcia Cecília M. F. Rubira

Institute of Computing – University of Campinas (UNICAMP) – Brazil
{afgarcia, cmrubira}@dcc.unicamp.br

Abstract

Modern object-oriented software systems are getting more complex and have to cope with an increasing number of exceptional conditions to meet the system's dependability requirements. This work proposes a software architecture which integrates uniformly both concurrent and sequential exception handling. The exception handling architecture is independent of a specific programming language or exception handling mechanism, and its use can minimize the complexity caused by handling abnormal behavior. Our architecture provides during the first design stage the context in which more detailed design decisions are made in later design stages related to exception handling. In this sense, this work also presents a set of design patterns which describes the structural and behavioral aspects of the components of the proposed software architecture.

1 Introduction

Modern object-oriented software systems are getting more complex and have to cope with an increasing number of error conditions to meet the system's dependability requirements. Dependable object-oriented software detects errors caused by residual faults and employs fault tolerance measures to restore normal computation [3]. Exception and exception handling provide a suitable scheme to detect and handle errors, and also incorporate fault tolerance activities into software systems. The detection of an error will result in an exception being raised, with an appropriate handler corresponding to the raised exception being automatically invoked to implement the fault tolerance measures [3]. The presence of *exception handling facilities* can reduce software development efforts since they allow software designers to: (i) represent errors as exceptions, (ii) define handlers to deal with them, and (iii) use an adequate strategy for exception handling when the occurrence of an exception is detected.

Moreover, object-oriented systems may be consisted of various execution threads (or processes) executing methods concurrently on objects. Exceptions are more difficult to handle and exception handling facilities to provide in concurrent object-oriented systems than in sequential ones specially because of cooperative concurrency [2]. That is, several threads usually cooperate to perform some *concurrent cooperative activity* of the system, giving rise to very complex concurrent interactions. In this context, erroneous information may be spread directly or indirectly through inter-thread communication during a cooperative activity. A general approach for structuring cooperative activities and employing exception handling in concurrent systems extends the well-known atomic action notion [2]. An atomic action is formed by a group of participants which are executed by cooperating threads. The group cooperate within the scope of an action and complex interactions are coordinated by the action, including the management activities related to concurrent exception handling. Participants may join an action asynchronously but they have to leave it synchronously to guarantee that no information is smuggled to or from the action. When an exception is raised in any of the participants inside an action, all action participants should participate in the error handling [2]. In general, different handlers for a same exception have to be called in the participants. These handlers are executed concurrently in order to handle the exception in a coordinated way. An additional difficulty is that several exceptions can be raised concurrently by participants during a cooperative activity. In this situation, a process of exception resolution is required to agree on the exception that should be notified to all participants.

Exception handling facilities for sequential programs are usually incorporated in various modern object-oriented programming languages, such as C++, Java and Eiffel. However, very few languages give direct support to concurrent exception handling (for instance, Arche); but, in general, the solutions presented cope with concurrent exception handling in a rather limited form. Recently some 'ad hoc' solutions have been proposed to the provision of concurrent exception handling which extends programming languages, such as Ada and Java [5, 8]. However, we believe that these recent proposals present complex solutions which are also very language-dependent and error prone. Besides, these solutions can be very intrusive from the viewpoint of the application since its normal code is usually amalgamated with explicit references and invocations of procedures responsible for exception resolution and final synchronization of the action participants. In addition, the task of software developers is also complicated in the sense that they have to implement exception resolution functions for each concurrent cooperative activity of the system. Consequently, these solutions present exception handling techniques which are difficult for software developers to use, and may produce software products which are non-reliable and difficult to understand, maintain and reuse.

The present interest in software architectures and design reuse motivated us to develop an exception handling software architecture for building robust software. The proposed architecture provides a generic infrastructure which supports uniformly both concurrent and sequential exception handling. Moreover, the exception handling architecture is independent of a specific programming language or exception handling mechanism, and its use can minimize the complexity caused by handling abnormal behavior. Our architecture provides during the first design stage the context in which more detailed design decisions are made in later design stages related to exception handling. A software system's quality requirements (or attributes) are largely permitted or restrained by its architecture; so if an appropriate architecture is chosen since the outset of the design phase, a proper use of exception handling throughout the development life cycle of a system can be obtained. The architecture is composed of four well-defined components: (i) the *Exception* component, (ii) the *Handler* component, (iii) the *Exception Handling Strategy* component, and (iv) the *Concurrent Exception Handling Action* component. The structural and behavioral aspects of the components are described by means of a set of design patterns. The patterns follow the overall structure of the *Reflection* architectural pattern which allows a clear and transparent separation of concerns between the application's functionality and the exception handling facilities, easing the task of building robust software.

2 Background

2.1 Software Architecture and Patterns

A system's software architecture abstractly describes the system's gross organization in terms of components and their interrelationships [6]. Components are physical and replaceable parts of a architecture, and serve as the building blocks for the architecture of a system. To each component are attached responsibilities. The components must interact with each other in the prescribed fashion, and each component must fulfill its responsibilities to the other components as dictated by the architecture. Each component conforms to and provides the realization of a set of interfaces.

Software patterns are an important vehicle for constructing high-quality architectures. Patterns are useful mental building-blocks for dealing with limited and specific design aspects when developing a software architecture [1]. *Architectural patterns* define the basic structure of an architecture and the systems implementing that architecture. *Design patterns* are however more problem-oriented than architectural patterns, and are applied in later design stages.

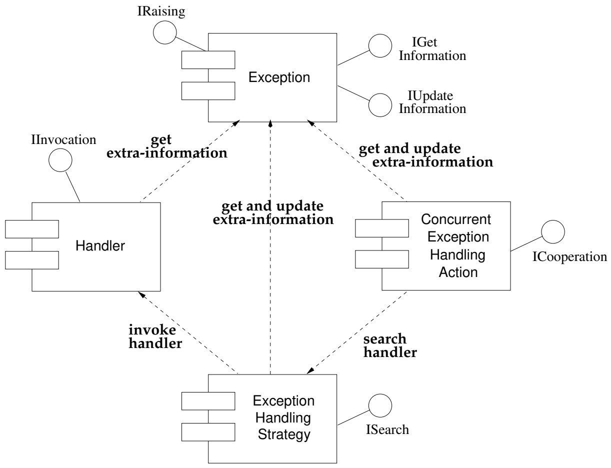

Figure 1: The Software Architecture for Exception Handling

2.2 Computational Reflection and Meta-Level Architectures

Computational reflection is a technique that allows a system to maintain information about itself (*meta-information*) and use this information to adapt its behavior [4]. In the object model, computational reflection establishes a *meta-level architecture* which achieves a *separation of concerns* between applications and management mechanisms. Meta-level architectures are composed of at least two dimensions: (i) a *base level* (or application level), and (ii) a *meta-level* (management level). The base level encompasses the objects responsible for implementing the functionality of the application. The meta-level encompasses the objects that deal with the processing of meta-information and management of an application. The meta-level objects (*meta-objects*) are associated with application objects, and maintain structural and behavioral information about the associated base-level objects. Meta-objects can intercept, inspect and modify transparently the effects of operations of the application objects. A meta-object protocol (*MOP*) establishes an interface of communication between base-level and meta-level objects.

3 The Architecture for Exception Handling

3.1 The Basic Architecture

This section presents a generic software architecture that integrates sequential and concurrent exception handling (Figure 1). Applications reuse our architecture to handle their exceptional situations by using the exception handling facilities provided by the architecture's components. The architecture is composed of four components: (i) the *Exception* component, (ii) the *Handler* component, (iii) the *Exception Handling Strategy* component, and (iv) the *Concurrent Exception Handling Action* component. Table 3.1 summarizes the components and their responsibilities. The responsibilities are classified into two kinds: (i) *application-dependent* responsibilities (ADR), and (ii) *application-independent* responsibilities (AIR). Application-dependent responsibilities are directly related to the application's functionality and include, for instance, facilities for specification of exceptions and handlers, raising of application exceptions, and specification of concurrent cooperative activities. The achievement of these responsibilities is application-dependent. As a consequence, the architecture's components provide the application developers with appropriate support in order to fulfill their application-dependent responsibilities. Developers of applications either invoke services provided by the architecture's component interfaces (Figure 1), or else refine the design of architecture's components according to their needs (Section 3.2).

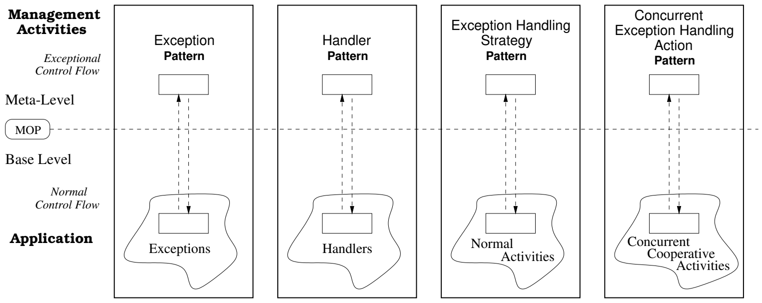

Figure 2: The Architecture Refinement

#	Component	Responsibilities
1	Exception	Specification and raising of exceptions (ADR) Management of extra-information (AIR)
2	Handler	Specification of handlers (ADR) Invocation of handlers (AIR)
3	Exception Handling Strategy	Search of handlers (AIR) Deviation of the control flow (AIR)
4	Concurrent Exception Handling Action	Specification of concurrent cooperative activities (ADR) Synchronization and exception resolution (AIR)

Table 3.1: Components and their Responsibilities.

Application-independent responsibilities include, for instance, facilities for extra-information management, handler invocation, deviation of the control flow, search of handler, synchronization and exception resolution. These responsibilities are related to management activities of exception handling. Components of our proposed architecture perform their management activities in a way that is transparent to the application (Section 3.2).

3.2 The Architecture Refinement

As stated previously, software designers reuse the architecture and tailor the components of the proposed architecture to add the functionality related to specific applications. Note that each architectural component may include application's functionality and management activities for exception handling. In order to obtain a clear separation of concerns between the application's functionality and the exception handling services, the architecture and their components incorporate a meta-level architecture, following the overall structure of the *Reflection* pattern [1]. This pattern captures the benefits of computational reflection and meta-level architectures (Section 2.2). Figure 2 presents the proposed meta-level architecture which is composed of two dimensions: the base level, and the meta-level.

Design patterns are proposed in order to refine the general components of the proposed architecture, providing the detailed design solutions. Each proposed design pattern presents the solutions for the specific design problem of the corresponding component. The *Exception* design pattern allows to separate the objects representing the exceptions of the application (base level) from the objects responsible for managing information about exceptions (meta-level). Meta-objects are associated transparently with base-level exception objects for keeping meta-information about their corresponding exceptions. The *Handler* design pattern allows to separate the objects containing the application's handlers (base level) from the objects responsible for invoking transparently these handlers at run-time (meta-level). After an exception is raised and the normal control flow is interrupted by the MOP, the meta-object invokes transparently the appropriate handler for that exception. The *Exception Handling Strategy* design pattern allows to separate the objects implementing the application's normal activities (base level) from the objects responsible for implementing the exception handling strategy (meta-level). Results of methods are checked transparently by the meta-objects in order to verify if such methods have raised any exception. MOP intercepts at run-time the exceptional results and deviates the normal control flow of the base-level application to the exceptional one at the meta-level. Following deviation an eligible handler is sought and executed. After management activities related to exception handling are finished, MOP returns the computation to the application's normal flow. The *Concurrent Exception Handling Action* design pattern allows to separate the objects implementing the application's cooperative activities (base level) from the objects responsible for implementing the participant synchronization and exception resolution (meta-level). This pattern performs the synchronization of the participants, and the resolution process in a way that is transparent to the application.

4 Related Work

The paper [5] describes a scheme for concurrent exception handling based on atomic action structures for the Ada95 language. In this approach, application's programmers have to implement a exception resolution function for each cooperative activity. Application code is also intermingled with invocations of *head processes* which are procedures for synchronizing the participants while exiting from an atomic action. In this way, application objects are polluted with explicit references for *head processes* which in turn reduce readability, maintainability and reusability of the applications. The coordinated atomic action concept [7] was introduced as a unified approach for structuring complex concurrent activities and supporting error handling between multiple interacting objects in a concurrent object-oriented system. Zorzo *et al.* [8] had developed a object-oriented scheme for implementing coordinated atomic actions within the Java language. In this scheme, application's programmers are provided with a number of classes to structure their application. However, such solution proposes a very simple exception handling mechanism. There is a single method intended to handle the exceptions raised during a cooperative activity. The structure of such a handler is very complex since a single handler must incorporate measures for handling all exceptions of the activity.

5 Conclusions

Designers of exception handling mechanisms have not paid enough attention to the integration of concurrent and sequential exception handling. The proposed software architecture in this work supports uniformly concurrent and sequential exception handling. Our architecture provides during the first design stage the context in which more detailed design decisions are made in later design stages related to exception handling. A software system's quality attributes are largely restrained by its architecture; so if an appropriate architecture is chosen since the outset of the design phase, a proper use of exception handling throughout the development life cycle of a system can be obtained. In this context, the set of proposed design patterns are used to describe the structural and dynamic aspects of the components of the proposed architecture. The design patterns incorporate a meta-level architecture that achieves a clear and transparent separation between the application's functionality and the management activities related to exception handling. This explicit separation of concerns facilitates the construction of robust applications and improves their readability, maintainability and reusability. Specific applications reuse the exception handling facilities provided by the proposed architecture's components, and their developers may concentrate on the application-dependent functionality.

References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns: Patterns-Oriented Software.* John Wiley & Sons, 1996.

[2] R.H. Campbell and B. Randell. Error Recovery in Asynchronous Systems. *IEEE Transactions on Software Engineering*, SE-12(8):811–826, August 1986.

[3] P.A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice.* Springer-Verlag, 2 edition, 1990.

[4] P. Maes. Concepts and experiments in computacional reflection. *ACM SIGPLAN Notices*, 22(12):147–155, December 1987.

[5] A. Romanovsky. Practical Exception Handling and Resolution in Concurrent Programs. *Computer Languages*, 23(7):43–58, 1997.

[6] M. Shaw and D. Garlan. *Software Architecture - Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[7] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, and Z. Wu. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. In *FTCS-25: 25th International Symposium on Fault Tolerant Computing*, pages 499–509, Pasadena, California, 1995.

[8] A.F. Zorzo, A. Romanovsky, J. Xu, B. Randell, R.J. Stroud, and I.S. Welch. Using Coordinated Atomic Actions to Design Dependable Distributed Object Systems. In *Workshop of Dependable Distributed Object Systems - OOPSLA'97*, pages 241–259, Atlanta, October 1997.