Structuring Exception Handling for Dependable Component-Based Software Systems

Paulo Asterio de C. Guerra    Fernando Castor Filho    Vinicius Asta Pagano
Cecília Mary F. Rubira
Institute of Computing - Campinas State University
P.O. Box 6176. CEP 13083-970, Campinas, SP, Brazil.
{asterio, fernando, cmrubira}@ic.unicamp.br, vinicius.pagano@eldorado.org.br
+55 (19) 3788-5842 (phone/fax)

Abstract

Component-based development (CBD) is recognized today as the standard paradigm for structuring large software systems. However, the most popular component models, such as Enterprise JavaBeans and DCOM, provide almost no guidance about how to incorporate exception handling into component-based systems. The problem of how to employ language-level exception handling mechanisms to incorporate fault-tolerance in component-based systems is recognized by CBD practitioners as very difficult and often not adequately solved, with negative impact on reliability and maintainability. We believe that the construction of dependable component-based systems requires both (i) a global exception handling strategy for inter-component composition and (ii) a local exception handling strategy for dealing with errors within the reusable components. In this paper we describe a general strategy for the development of fault-tolerant software components and their integration into dependable component-based software systems.

1. Introduction

Component-based development (CBD) [27] is employed today to build large software systems with high dependability requirements, such as e-commerce and e-banking. The central tenet of CBD is that software should be developed by integrating pre-existing reusable software components, usually built by different organizations. A direct implication of this notion is the division of responsibilities between component development and system integration. On the one hand, developers of reusable software components do not have full knowledge of the different operational environments in which the components will be integrated. On the other hand, these components are usually provided as 'black-boxes', with their internal design and source code kept inaccessible to systems integrators.

The separation of component development and system integration imposes limitations on the use of traditional software assurance technologies for component-based software systems [29]. Since few assumptions can be made about the reliability of third-party components integrated in a new system, CBD requires approaches to fault tolerance that take into account the particularities of this new development paradigm.

Exception handling[6] is a well-known technique for incorporating fault tolerance into software systems. Many modern programming languages, such as C++, Java, and C#, provide sophisticated exception handling systems (EHS) to allow the structuring of the exceptional activity of a software system. However, even in systems that are not component-based, these EHS are often misused and/or abused [20]. The proper use of an EHS requires a consistent, global strategy for defining exception types and allocating responsibilities among exception handlers. Designing such exceptional strategy is seen as a difficult task for application developers, who tend to concentrate in the design of normal activity, related to the functional requirements of applications. Structuring exception handling is even more difficult for developers of reusable software components that are not aware of the contexts in which these components will be used, and for system integrators, who should consider the local exception handling strategies of the components to be integrated.

One possible approach for building fault-tolerant component-based systems is to employ exception handling at the architectural level, as suggested by some authors in the literature [3, 11, 13]. However, exception handling at the architectural level is not a replacement for implementation-level exception handling [3, 13]. The two techniques are complementary and should be employed synergetically in
order to achieve the best results. Nevertheless, to the best of our knowledge, no attempts have previously been made to devise a general strategy for structuring component-based systems which takes into account both architectural-level and implementation-level exception handling.

When integrating software components to build dependable systems, it is of critical importance to resolve conflicts between the exception handling policies of the components and of the architecture of the system. If these conflicts are not solved, there are at least two possible undesirable consequences of an exception being raised by a component: (i) the context and semantics of the exception are lost, making it impossible for other components to handle it, and (ii) the exception may simply be ignored, leading to the propagation of errors throughout the system. Our practical experience in component-based mentoring for various Brazilian companies has shown us that this is a recurring problem and motivated us to devise a general exception handling approach for component-based software systems.

In this paper, we propose a strategy for exception handling in dependable component-based software systems that is based on a generic exception type hierarchy and the definition of different kinds of exception handlers. The generic exception type hierarchy defines abstract exception types with precise semantics that are used to map an exception raised by a component to a generic type known by the global exception handling strategy. The different kinds of handlers allow the division of exception handling activities between individual components (implementation-level) and connectors (architectural-level). Connectors are architectural elements that mediate the interactions between components in a specific system configuration. We also define a set of guidelines to help in: (i) the development of software components that are both reusable and robust; and (ii) the instantiation of the proposed strategy to a particular software system.

Our ultimate goal is to provide to component developers and system integrators a set of design and implementation directives that allows them to better structure the systems they build. In this manner, the impact of fault tolerance mechanisms on the overall system complexity is reduced and systems that are both more reliable and easier to maintain can be produced.

The rest of this paper is organized as follows. Sections 2 and 3 provide some background on exception handling and software architecture [2], respectively. Section 4 presents the strategy for exception handling from the perspective of both system integrators and component developers. In Section 5, we describe some of the lessons learned from an initial assessment. Section 6 presents some related work and Section 7 concludes the paper.

2. Fault Tolerance and Exception Handling

The complexity introduced by fault tolerance in software systems motivated the development of a well-known style of system design known as idealised fault-tolerant component (IFTC) [1]. An idealised fault-tolerant component is a piece of software (a class, module, component, or a whole system) where the parts responsible for normal and abnormal activities are separated and well-defined. Figure 1 presents the structure and flow of control of the IFTC. Upon the receipt of a service request, an IFTC produces three types of responses: normal responses in case the request is successfully processed, interface exceptions in case the request is not valid, and failure exceptions, which are produced when a valid request is received but cannot be successfully processed.

Exception handling is a very popular technique for incorporating fault tolerance into software systems. An exception handling system allows software developers to define exceptional conditions and to structure the abnormal activity of software components. When an exception is raised, the underlying EHS interrupts the normal processing and transfers the control to an appropriate exception handler to deal with the exceptional condition. Handling contexts are regions in which the same exception types are treated in the same way.

In [6], Flaviu Cristian presents a synthesis of the termination exception handling paradigm for sequential programs. The EHS of C++, Java, and C# adhere to this model of exception handling. The Design by Contract approach [17] provides a slightly different view of this paradigm for exception handling, supported by the Eiffel language and development environment. In the remainder of this section we briefly examine these two approaches and discuss their main differences.

The main focus of the approach of Cristian is robustness, which is a means to fault tolerance. A robust program should be prepared to handle all possible inputs, in confor-
mance to a specification. A program specification defines a standard exit point and zero or more exceptional exit points. An exceptional exit point corresponds to an abnormal condition that is anticipated by the designers. A program may terminate normally, at its standard exit point, or exceptionally, at one of its declared exceptional exit points. In the second case, an exception should be signalled. There may also be undeclared exceptions that are used to signal program failures, which result from unanticipated abnormal conditions (or design faults).

The main goal of the Design by Contract approach is correctness, that is, it focuses on avoiding faults, not tolerating them. A routine should not be prepared to handle all possible inputs, but only those specified by the pre-condition of its contract. A routine has a single contract that specifies a single exit point. This exit point is taken whenever the routine succeeds to fulfill its contract. Exceptions are only used to signal design faults, which are detected by means of executable assertions that describe the contracts.

4. Proposed Exception Handling Strategy

We assume that the specification of a component, or simply specification, may include exceptional specifications to define expected behaviour when some abnormal, but anticipated, exceptional conditions occurs. The exceptional specification associates anticipated exceptional conditions with a number of declared exceptional exit points. The semantics of a declared exceptional exit point is defined by the specification and is part of the contract with clients of the component. Any correct implementation of a specification should include detection of anticipated exceptional conditions. However, a more robust implementation of a specification may include the detection of exceptional occurrences that are not anticipated by the specification. For these unanticipated exceptional conditions, the developer should define new undeclared exceptional exit points.

Undeclared exit points are problematic because different correct implementations of the same specification may define different undeclared exit points. This may result in architectural mismatches [10] occurring when trying to integrate such components in a system. It is a current practice to associate undeclared exit points with exceptions of arbitrary types that are defined by the implementation of the component or are propagated from lower level components. This "ad hoc" scheme for signalling unanticipated exceptional conditions results in a large number of undeclared exception types which are meaningless to system integrators. Without the proper contextual information and failure semantics, there is little opportunity for introducing fault tolerance in the integrated system.

Our overall strategy to solve this problem comprises two complementary facets: a global, or inter-component, strategy and a local, or intra-component, strategy. The inter-component strategy is concerned with system integration and is applied to configurations of components and connectors. The intra-component strategy is concerned with component development and is applied to individual reusable software components. To allow these two strategies to be applied in conjunction, they share a common abstract exception type hierarchy for precisely expressing the failure semantics of a component or connector. The following subsections describe the abstract exception type hierarchy, the inter-component strategy, and the intra-component strategy.

4.1. Abstract Exception Type Hierarchy

Figure 2 shows the proposed abstract exception type hierarchy. On the top of the hierarchy is Exception, the superclass of all exception classes. This hierarchy is easily mapped to existing object-oriented programming languages, such as Java and C#. A component terminates at a declared exceptional exit point by signalling an exception
of the abstract type DeclaredException. All the exceptions of type DeclaredException, as well as its subtypes, should be explicitly declared in the signatures of operations that may signal them, if this feature is supported by the underlying programming language.

The UndeclaredException hierarchy is used by component developers to attach implementation-specific failure semantics to undeclared exceptions. These abstract exception types also allow system integrators to incorporate in a component-based system handlers to deal with unanticipated exceptions in a systematic way. UndeclaredException has two direct subtypes: RejectedRequestException and FailureException. Exceptions of the RejectedRequestException type are used to signal that a request received from a client could not be processed, due to a precondition violation, and that the state of the system was not affected.

Exceptions of the type FailureException indicate that the implementation of the component failed to process a valid request. FailureException has two subtypes: RecoveredFailureException and UnrecoveredFailureException. Exceptions of the RecoveredFailureException type are used to indicate that a request received from a client could not be processed, due to a precondition violation, and that the state of the system was not affected.

The following directives provide additional guidance as to how exceptions should be translated by an architectural connector. They are based on the configuration depicted in Figure 3 and on the premise that the Server component signalled an exception E1 upon receipt of a request sent by the Client component.

- If E1 is declared in both the required interface of the Client component and the provided interface of the Server component, E1 may be automatically propa-
gated;

- If E1 is declared in the provided interface of the Server component and there is a corresponding exception type E2 (compatible semantics) declared in the required interface of the Client component, an exception of type E2 is propagated. A typical example of this scenario occurs when exceptions E1 and E2 have a common ancestor, E3: Another example occurs when E2 is a supertype of E1;

- If E1 is declared in the provided interface of the Server component and can not be translated according to the two rules stated above, then three different outcomes are possible:
  - If E1 was signalled because the request received by the Server component was not valid, and the state of the latter was not changed, an exception of type RecoveredFailureException should be signalled;
  - If E1 was signalled because the Server component received a valid request and was unable to process it, if the state of the component has not been damaged, a subtype of RecoveredFailureException is signalled;
  - If E1 was signalled because the state of the Server component has been damaged somehow, either due to an invalid request of due to a valid request that could not be processed, a subtype of UnrecoveredFailureException should be signalled;

- If E1 is a subtype of UndeclaredException, it may be automatically propagated;

- If E1 is not declared in the provided interface of the Server component and is not a subtype of UndeclaredException, the propagated exception should be of an instance of UndeclaredException, indicating that the Server component may be in an inconsistent state.

4.3. Intra-Component Exception Handling Strategy

The intra-component exception strategy deals with the development of reusable software components and is based on application-level exception (ALE) handlers. ALE handlers are associated to the implementation classes of the component, that is, the classes that implement its provided operations, and to its fade classes. A façade [9] defines an access point to the implementation classes associated to a given provided interface and serialises all incoming requests, in order to transform the component in a damage confinement region [1]. The main responsibility of an ALE handler is to deal with anticipated exceptions, which are more related to the application domain of the component and are declared in its required interfaces.

Besides implementing the normal activity behaviour of the component, the implementation classes are also responsible for: (i) detecting exceptional conditions anticipated by the specification and signalling them by raising internal exceptions of types declared in the provided interface of the component; (ii) signalling other exceptional conditions which are specific to the implementation of the component, by raising internal exceptions; and (iii) executing clean-up actions systematically in every method, independently of whether an exception occurs or not. Java provides explicit support for this by means of finally blocks. The types of internal exceptions raised by the implementation classes of components may be subtypes of both UndeclaredException and DeclaredException, depending on the type of error that occurred.

ALE handlers are responsible for handling three kinds of exceptions: (i) external exceptions of types declared in the required interfaces of the components; (ii) internal exceptions raised by the implementation classes of the component; and (iii) internal exceptions raised by the underlying infrastructure. When possible, ALE handlers should implement forward error recovery to mask these three kinds of exceptions.

ALE handlers are also responsible for dealing with exceptions that reach the boundary (one of the façades) of the component. Exceptions of types declared in the provided interface of the component and exceptions of type RejectedRequestException are simply propagated to the client. For other exception types, backward error recovery should be performed, in case it is available. If it is successful, an exception of type RecoveredFailureException is signalled, indicating that the state of the component is consistent. Otherwise, and exception of type UnrecoveredFail-
ureException is signalled.

If components and connectors should not or cannot be modified in order to introduce ALE handlers, a wrapper should be created and the appropriate handlers associated to it. Wrappers allow developers to alter the externally visible behaviour of components that could not be otherwise modified. They are used for a wide range of applications and are a very important element of software development based on off-the-shelf components [25].

4.4. Applying the Guidelines

We have devised a simple set of steps that help in the application of the guidelines described in this section. These steps aim to make the use of the guidelines an incremental and intuitive activity and should be used in conjunction with a component-based development methodology [4, 21]. They are based on our previous experience in the use of exception handling for building fault-tolerant component-based systems [12, 21]. We assume that, before applying these steps, some analysis and design activities have been performed and failure models of the components and of the architecture have been defined.

The following steps should be performed, in order to apply the proposed guidelines:

1. First, implement the subtypes of the exceptions specified by the abstract exception type hierarchy (Section 4.1), if necessary. These exceptions are dependent on the application and on the types of errors expected.

2. Implement the ALE handlers. In Java, this activity is usually performed using try-catch blocks. In case the source code for the components is not available or should not be modified, a wrapper must be created to encapsulate exception handling activities. Either way, we suggest that the actual handlers be implemented as methods in separate classes responsible exclusively for exception handling. In this manner, normal and exceptional behavior are more explicitly decoupled [21] and exception handlers can be reused.

3. Finally, CLE handlers should be implemented. These should be capable of handling all the undeclared exceptions that may be signaled by server components (Figure 3 in Section 4.2). We believe connectors are the best places for introducing generic exception handlers, that is, handlers capable of dealing with exceptions of any type. This is so because, as mentioned earlier, component developers cannot foresee the contexts in which the components they build will be used. In order for generic handlers to be useful, they must be capable of taking reasonable actions independently of the type of exception received. We believe that this is only possible if knowledge about the application as a whole is available.

5. Evaluation

Until the present moment, we have performed two case studies to evaluate the ideas presented in this paper. These case studies have demonstrated the feasibility of the proposed approach and pointed some limitations that should be addressed in the future. The first case study was developed by the third author during his master’s work. The other is an ongoing work being developed by the first author, together with a Brazilian company. In this section we briefly describe the first case study and some of the lessons learned until the present moment.

We have applied the proposed guidelines to a real-world software system, called Telestrada. Telestrada is a large traveler information system being developed for a Brazilian national highway administrator. It comprises five subsystems: Central Database Subsystem, GIS (Geographic Information System) Subsystem, Call-Center Operations Subsystem, Roadside Operations Subsystem, and Complaint Management Subsystem. The case study consisted of applying the guidelines presented in this paper to the Complaint Management subsystem, in order to model its exceptional behavior. This subsystem is a web-based application implemented in Java using the COSMOS implementation model [24]. The main goal of this case study was to evaluate the impact of applying these guidelines to an existing system, both in terms of separation of concerns between normal and abnormal activity and in terms of modifications made to the implementations of the components.

The application of the presented guidelines to the Complaint Management subsystem produced an implementation in which an explicit separation between normal and exceptional behavior could be achieved. Although this separation is not complete, in the sense that catch blocks are still present in the normal application code, all the code responsible for the actual exception handling procedures could be put into separate classes. In this manner, exception handlers can be modified without the need to affect the normal behavior of the system. Moreover, this separation allows handlers to be reused. This is an important feature since the number of places where a given type of exception may be signaled is usually much higher than the number of ways in which it can be handled [15].

Modifications applied to existing classes in order to adapt their exception handling strategies to the one we propose required moving approximately 5% of the application code to handler classes (from a total of almost 9000 lines of code). This task imposed a time overhead of approximately two programmer/hours, which we found acceptable. The availability of free code restructuring tools, such as the
refactoring features of the Eclipse IDE [8] greatly helped reducing the total time required for the task. The implementation of CLE handlers, which were built from scratch, also imposed little time overhead, since the system configuration comprises just six connectors and their handling policies consisted of translating exceptions between different conceptual domains.

Changing the exception type hierarchy of the application in order to conform to the one presented in Section 4.1 required a fairly large amount of additional work. This happened due to the great number of places in the code where exceptions were raised or handled, that had to be reviewed (more than 250). We think the overhead would be much reduced if the guidelines had been followed since the beginning of development, but we have not yet collected data in order to support this claim.

The idea of incorporating the use of wrappers in the proposed guidelines arose from a situation in which we had to adapt the behaviour of an architectural connector whose source code could not be modified. The addition of CLE handlers to this connector, by means of the wrapper, avoided an error that kept happening systematically. Other components and connectors in the configuration remained oblivious to the change.

Finally, the developers of both case studies provided some nice feedback on the applicability of the proposed approach. They found the division of responsibilities very useful and intuitive, and suggested some modifications that have already been incorporated in the guidelines. Although this data is not empirical and cannot be taken into account to determine the general applicability of the guidelines, we think it is very useful as the result of an initial assessment.

6. Related Work

Software fault-tolerance at the architectural level is a young research area that has recently gained considerable attention. Some approaches based on the idea of design diversity [1] have been developed in the context of the reliable evolution of component-based distributed systems. Both the Hercules framework [5] and the concept of Multi-Versioning Connectors [19] maintain old and new versions of components working concurrently, in order to guarantee that the expected service is provided, even if there are faults in the new versions. The guidelines described in Section 4.2 for handling exceptions at connectors build upon these two approaches.

Castor et al [3] have proposed an EHS addressing the specific concerns of component-based systems, at the architectural level. This work does not address issues specific to implementation-level exception handling, neither its integration with architectural-level exception handling.

In a similar vein is the work by Issarny and Banâtre [13], which describes an extension to existing architecture description languages [16] for specifying architectural-level exceptions (configuration exceptions). This work emphasizes fault treatment [1] at the architectural level, by means of architecture reconfiguration.

Saridakis and Issarny [22] focus on the formal description of architectures in order to prove they are reliable. By employing these specifications together with refinement laws which guarantee the preservation of the reliability property, the authors intend on producing concrete architectural descriptions which are easily translated to code.

7. Conclusion

The main contribution of this paper is a general strategy for exception handling in component-based systems, addressing the problem of how do develop robust and reusable software components that can be easily integrated in dependable component-based systems. We have drawn ideas from different views on exception handling [6, 17] and combined them in a set of guidelines for structuring exception handling at both the architectural and the implementation levels.

The strategy is presented using the COSMOS model to explicitly materialize architectural elements at the implementation level. In spite of this, we believe it is possible to use it with well-known component standards, such as Enterprise JavaBeans [26] and DCOM [18]. Exception handling in these standards is based on the exception handling systems of the underlying programming languages. Hence, in order to apply the guidelines suggested in this paper to a system based on Enterprise JavaBeans or DCOM components, all that is required is a mapping from COSMOS to these standards. One such mapping is described elsewhere [7].

One limitation of the proposed approach is that it is not suitable for programming languages that do not provide an exception handling system. For instance, it is not trivial to map our abstract exception type hierarchy (Section 4.1) to the possible results of a C function that returns an integer.

Our most immediate future work consists of the development of tools for partially automating the implementation of handlers at both inter-component and intra-component levels. This is an ongoing work that is being conducted in the context of a larger project involving both the Software Engineering and Fault Tolerance group [28] at the IC/UNICAMP, and a Brazilian company.

Furthermore, we also intend to evaluate the applicability of aspect-oriented programming [14] (AOP) techniques to help in decoupling the implementation of the normal and abnormal behaviours of systems built according to the proposed guidelines. Until the present moment, work on the subject has focused exclusively on reducing the amount of
code responsible for exception handling [15].

Acknowledgements

We would like to thank the anonymous referees, who provided many interesting comments and suggestions. Fernando is supported by FAPESP/Brazil, grant number 02/13996-2. Cecilia is partially supported by CNPq/Brazil, grant number 351592/97-0.

References