A Systematic Approach for Structuring Exception Handling in Robust Component-Based Software

**Fernando Castor Filho, Paulo Asterio de C. Guerra,
Vinicius Asta Pagano and Cecília Mary F. Rubira**

Institute of Computing - State University of Campinas (UNICAMP)
P.O. Box 6176. CEP 13084-971, Campinas, SP, Brazil.
{fernando, asterio, cmrubira}@ic.unicamp.br, vinicius.pagano@eldorado.org.br

Abstract

Component-based development (CBD) is recognized today as the standard paradigm for structuring large software systems. However, the most popular component models and component-based development processes provide little guidance on how to systematically incorporate exception handling into component-based systems. The problem of how to employ language-level exception handling mechanisms to introduce redundancy in component-based systems is recognized by CBD practitioners as very difficult and often not adequately solved. As a consequence, the implementation of the redundant exceptional behaviour causes a negative impact, instead of a positive one, on system and maintainability. In this paper, we propose an approach for the construction of dependable component-based systems that integrates two complementary strategies: (i) a global exception handling strategy for inter-component composition and (ii) a local exception handling strategy for dealing with errors in reusable components. A case study illustrates the application of our approach to a real software system.

Keywords: Exceptional behaviour, Fault-tolerant component, Software fault tolerance, Component-based development.

1 Introduction

Component-based development (CBD)[36] is employed today to build large software systems, such as commercial and financial information systems with high dependability requirements. The central tenet of CBD is that software systems should be built by integrating pre-existing reusable software components, which may be developed by different organizations. A direct implication of this notion is the separation in time and space between component development and system integration. On the one hand, developers of reusable software components do not have full knowledge of the different contexts in which the components will be used. On the other hand, system integrators usually have limited access to the internal design and source code of these components. The construction/integration dichotomy leads to mismatches[13] between assumptions made by different components of an assembled system. Techniques for dealing with mismatches related to the functional properties of a system, such as wrappers and mediators[13], are in widespread use. However, mismatches related to conflicting assumptions regarding the behaviour of components when they deviate from their specifications (exceptional or abnormal behaviour) are not well understood. Failure to take the exceptional behaviour of components into account when building a component-based system compromises the analysability of the assembled system and its overall dependability.

Exception handling[8] is a well-known mechanism for introducing forward error recovery[1] in software systems. Many important object-oriented programming languages, such as Java, C++, and C# have incorporated this mechanism. In traditional software development, a large part of the code of a reliable software system is dedicated to detection and handling of exceptions[8]. However, this redundant part of the code is usually the least understood, tested, and documented. In component-based development, a similar phenomenon can be observed. Developers of large systems based on the J2EE platform[35], one of the de facto

standards in the industry for CBD, have habits concerning the use of exception handling that make applications more vulnerable to faults and harder to maintain[28].

The lack of systematic approaches for structuring the exceptional behaviour of component-based applications is an important factor that contributes to this situation. Existing component-based development processes, such as Catalysis[10] and UML Components[6], focus almost exclusively on the system's normal behaviour. There are some proposals in the literature for extending such processes with activities for designing the exceptional behaviour of component-based systems[2,29,40]. However, these proposals do not address the translation of the obtained design down to the implementation level of a component-based system. Also, the most popular component models, such as EJB (Enterprise Java Beans)[34] and .NET[24] rely almost entirely on the exception handling system (EHS) of the target programming language, providing little guidance about how to better incorporate exception handling into their component-based applications.

The proper use of an EHS requires a consistent strategy for defining exception types and allocating responsibilities to exception handlers. Structuring exception handling is even more difficult for developers of component-based systems, due to the construction/integration dichotomy as discussed earlier.

When integrating software components to build dependable systems, it is of critical importance to resolve conflicts between the exceptional behaviour of the reusable components and the intended exceptional behaviour of the assembled system. When these conflicts are not solved, they may result in undesirable situations, such as: (i) the context and/or semantics of an exception raised by a component are lost, making it difficult for other components to handle it; or (ii) an 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, in practice, 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 structuring exception handling in dependable component-based software systems. The proposed strategy is based on an abstract exception type hierarchy and the definition of different kinds of handlers with clearly pre-defined responsibilities. Component developers use the abstract exception type hierarchy to derive concrete exception types that preserve the semantics of a small set of generic exception types. These generic types are used by the system integrator to define an exception handling strategy for the integrated system that is not dependent of any particular implementation of its components. The different kinds of handlers promote separation of concerns between local (component-specific) and global (architectural) exception handling policies. The proposed strategy is based on two different and complementary views on exception handling. The first view is that presented by Flaviu Cristian in a classic article formally describing the termination model of exception handling in sequential programs[8]. The second is Bertrand Meyer's view, presented as part of the Design-by-Contract[20] methodology.

Our approach could be integrated within a typical component-based development process. The main requirement for this integration is the *a priori* execution of activities for defining the failure hypotheses of the system and designing the exceptional behaviour to be implemented. The execution of these tasks is not considered to be trivial and, in the literature, there are several works that address them[2,29,40]. We consider these works complementary to ours.

Our ultimate goal is to provide component developers and system integrators with a set of design and implementation guidelines that allows them to better structure the exceptional behaviour of the systems they build. In this manner, the impact of exceptional behaviour on the overall system complexity is reduced and the resulting system is both more reliable and easier to maintain. Furthermore, these guidelines should be easy enough to be applied to systems that do not have strict dependability requirements, and flexible enough to be used in conjunction with more sophisticated software fault tolerance mechanisms, such as design diversity[1].

The rest of this paper is organized as follows. Section 2 presents some related work, while Section 3 provides some background on exception handling, software architecture[30], and component-based development processes. 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 a real-world case study. Section 6 presents concluding remarks and ideas for future work.

2 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 reliable evolution of component-based distributed systems. Hercules framework[7] and Multi-Versioning Connectors[27] are approaches that 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 4.3 for handling exceptions at the architectural level are based on these two approaches.

Other 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[3,16,18]. The work by Issarny and Banâtre[18] describes an extension to existing architecture description languages[21] for specifying configuration exceptions, which are exceptions raised due to violations of architectural invariants. Guerra et al[16] have proposed an approach for architecting fault-tolerant component-based systems based on a specific architectural style. Castor et al[3] have proposed an EHS addressing specific concerns of component-based systems, at the architectural level, also focusing on a specific architectural style. These works differ from our present work in the sense that they do not attempt to integrate architectural-level and implementation-level exception handling. Architectural-level exception handling is not a replacement for implementation-level exception handling[3,18]. The two techniques are complementary and should be employed synergistically in order to achieve best results. To the best of our knowledge, however, no attempts have previously been made to devise a general strategy for structuring component-based systems taking into account both architectural-level and implementation-level exception handling.

There are some works in the literature describing guidelines for structuring exception handling in object-oriented software systems[9,32]. In general, these works do not focus on CBD and do not try to bridge the gap between architectural-level and implementation-level exception handling. In spite of this, they do provide valuable advice, which has been taken into account for the elaboration of the approach proposed in this paper.

More closely related to DBC, is the work of Shenoy[31] that discusses best practices in EJB exception handling. The main goal of Shenoy's work is faster problem resolution and it is based on the backward error recovery capabilities provided by EJB containers. In contrast, our main goal is a basis for forward error recovery and fault-tolerance that is not dependent on any specific component framework.

Vecellio[38] motivates the creation of techniques to assess the reliability of off-the-shelf (OTS) components. The author argues that traditional techniques for assuring the reliability of software systems are not effective for component-based systems. Meyer[23] reinforces these ideas and discusses the concept of trusted components. The author states that the elaboration of extensive techniques for demonstrating the quality of reusable components, together with the construction of a large set of trusted components, has the potential to change the way systems are developed. This viewpoint is complementary to ours.

Also related to our work, in the area of dependability benchmarking, it is possible to estimate the dependability of certain types of OTS components[19]. However, it is still difficult to predict how components built by different organizations will behave together when integrated into a new system.

3 Background

3.1 Exception Handling

The complexity introduced by fault tolerance in software systems motivated the development of a well-known style of system structuring 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 exceptional activities are separated and well defined. Figure 1 presents the structure and flow of control of the IFTC. Upon 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.

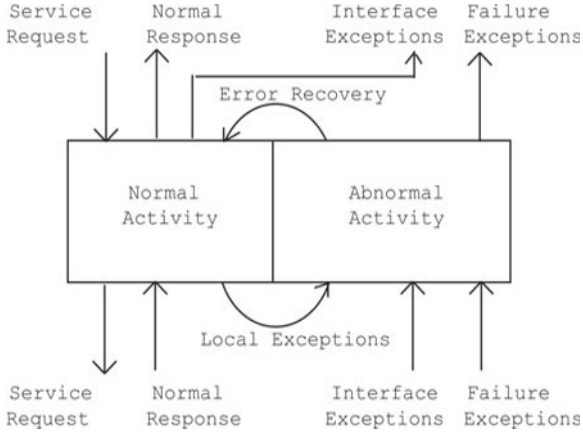

Figure 1. Idealised fault-tolerant component[1]

Exception handling is a very popular technique for incorporating fault tolerance into software systems. It allows developers to structure the redundant code that is added to deal with the *exceptional conditions* that may occur, separating it from the code responsible for the normal operating flow. An exceptional condition is signalled by means of an *exception* that is raised by the normal code. When this occurs, the underlying EHS interrupts the normal flow and transfers the control to an appropriate *exception handler*, which can deal with the exceptional conditions

associated with the *type of the exception* raised. Handling contexts are regions of the code in which exceptions of the same type are treated in the same way.

In [8], Flaviu Cristian presents a synthesis of the termination exception-handling paradigm for sequential programs. The exception handling systems of C++, Java, and C# adhere to this model of exception handling. The Design by Contract approach[22] provides a different view of exception handling, which is supported by the Eiffel language.

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

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 precondition 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 fulfil its contract. Exceptions are only used to signal design faults, which are detected by means of executable assertions that describe the contracts.

3.2 Software Architecture

The architecture of a software system shows how the system is realized by a collection of components and the interactions among them[30]. The building blocks of an architectural description are components, connectors, and architectural configurations. A component is a unit of computation or a data store. Therefore, components are loci of computation and state. Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions. Architectural configurations, or topologies, are connected graphs of components and connectors that describe architectural structures[21].

The realization of abstract software architectures requires concrete implementations, which raises the question about conformance of an implementation to the intended architecture. To be effective, solutions at the architectural level must be correctly mapped to the implementation level. It is not trivial to guarantee this conformance, since there is a semantic gap between the abstractions defined by software architecture, namely, architectural components and connectors, and the abstractions supported by mainstream object-oriented programming languages, such as packages and classes. In this work, we have used a component implementation model, called COSMOS[33], to bridge the gap between the software architecture of the system and its implementation.

The COSMOS model integrates a set of design patterns and guidelines into the implementation of a component-based system. These guidelines include: materialization of architectural elements at runtime; separation of non-functional concerns; clear separation between component's specification and implementation; explicit declaration of component's specification dependencies; strong encapsulation of implementation; separation of code inheritance from types hierarchy and loose coupling of implementation classes[33].

When using COSMOS, each architectural component is mapped, at the implementation level, to a package containing two sub packages: (i) the *specification package* contains the specification of the component's provided and required interfaces; and (ii) the *implementation package* contains the definition of the concrete classes that implement the component's behaviour. Architectural connectors are mapped to *connector packages* that implement the connections between required and provided interfaces of interacting components.

3.3 A Typical Component-Based Development Process

There are few CBD processes that have achieved some acceptance in the industry[6,10], compared to the large number of processes available for object-oriented development. This is not something to be surprised by. Although the first work proposing the use of "mass produced software components"[20] dates back to more than 30 years ago, most research on the subject has appeared in the last ten years.

Figure 2 presents a typical component-based development process divided into six workflows[6]: requirements, specification, provisioning, assembly (or integration), test, and deployment. The requirements workflow aims to identify the system requirements. The specification workflow structures the software architecture of the system as a set of abstract components that have specific responsibilities and interact to fulfil the system requirements.

The implementation of a component-based system is achieved by provisioning and assembly workflows. This is a consequence of the component development/system

integration dichotomy described in Section 1. Components are instantiated during the provisioning workflow. In this workflow, the system integrator decides if an abstract component can be instantiated by an existing OTS component, or if it will require an implementation effort, in which case it is called a newly developed component. The selection of an OTS component often requires its adaptation and, possibly, the refinement of the system's software architecture to fit the available OTS component. This adaptation may include the development of *wrappers*[13] to adapt the external interface of the OTS component to that specified for the abstract component being instantiated.

In the assembly workflow, the system integrator assembles OTS and newly developed components to build the whole system. This integration effort includes the development of glue code necessary to connect the various components, and comprises the specification and implementation of connectors and wrappers.

During the test workflow, the integrated system is tested and corrections may be made to ascertain that it fulfils its requirements and conforms to its specification. During the deployment workflow, the final system is installed in the environment of the user.

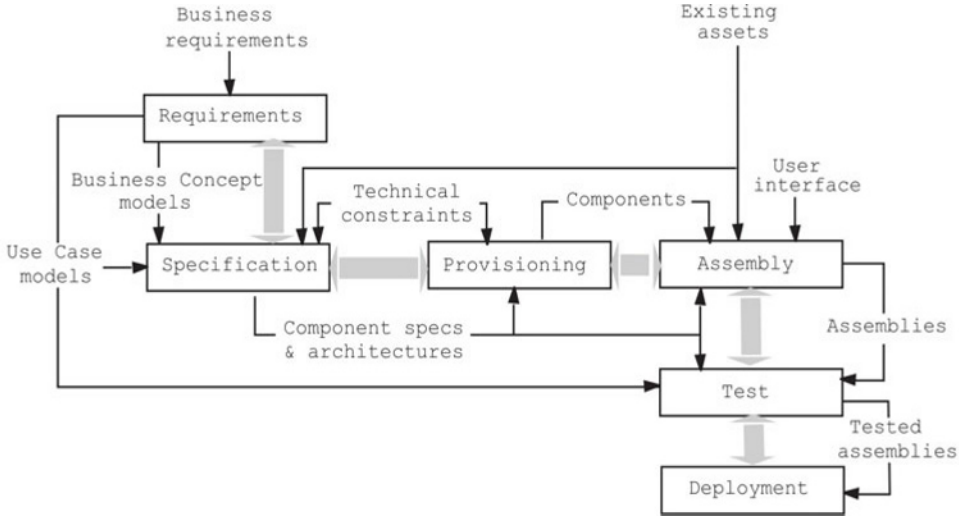

Figure 2. Workflows in the component-based development process[6]

4 The Proposed Exception Handling Strategy

We assume that the specification of a component includes its exceptional specification. The latter defines the expected behaviour when some abnormal, but anticipated, conditions occur. The exceptional specification associates anticipated exceptional conditions with a number of declared exceptional exit points (Section 3.1). A declared exceptional exit point, when taken, is signalled by an exception of a corresponding *declared exception type*. The semantics of a declared exception is defined by the specification and it is part of the component's interface specification. Any correct implementation of a specification should include detection of the anticipated exceptional conditions. However, a more robust implementation may include the detection of exceptional conditions that are not anticipated by the specification. For these unanticipated exceptional conditions, the component developer should define undeclared exceptional exit points.

Undeclared exceptional exit points are problematic because different correct implementations of the same specification may define different undeclared exceptional exit points. This may result in architectural mismatches[13] when one tries to integrate such components in a system. It is a current practice to associate undeclared exceptional exit points with exceptions of arbitrary types that are defined by the component developer or are propagated from lower level components. This *ad hoc* scheme for signalling unanticipated exceptional conditions may cause, during system execution, the raising of *undeclared exceptions* without proper contextual information and failure semantics. In these circumstances, the system integrator has little opportunity for introducing fault tolerance in the integrated system.

Our approach to solve this problem comprises two complementary strategies: a global (inter-component) strategy and a local (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.

4.1 Abstract Exception Type Hierarchy

Figure 3 shows the proposed abstract exception type hierarchy. This hierarchy can be easily mapped to existing object-oriented programming languages where exceptions are defined by classes, such as C++, Java, and Delphi. On the top of the hierarchy is `Exception`, the super class of all exception classes. A component's execution 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. The failure semantics associated with the abstract exception type hierarchy is based on the exception types defined for the idealized fault-tolerant component[1] and coordinated atomic actions[39].

The `UndeclaredException` hierarchy is used by a component developer to attach failure semantics to exceptions associated with exceptional conditions that are not anticipated by the component's specification. These abstract exception types also allow system integrators to incorporate handlers in a component-based system to deal with these undeclared 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 pre-condition violation, and that the system's state has not been 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 type `RecoveredFailureException` are used to indicate that, in spite of the fact that an error occurred, the component has been left in a consistent state. Instances of `UnrecoveredFailureException` are used to indicate that a failed operation may have caused undesirable effects in the state of the component.

4.2 Intra-Component Exception Handling Strategy

The intra-component strategy is applied during the provisioning workflow of a CBD process (Section 3.3). At this stage, concrete components are selected in order to materialize the abstract components specified in the software architecture of the system being developed. As discussed earlier, these components may be either existing or newly developed components. Our intra-component strategy

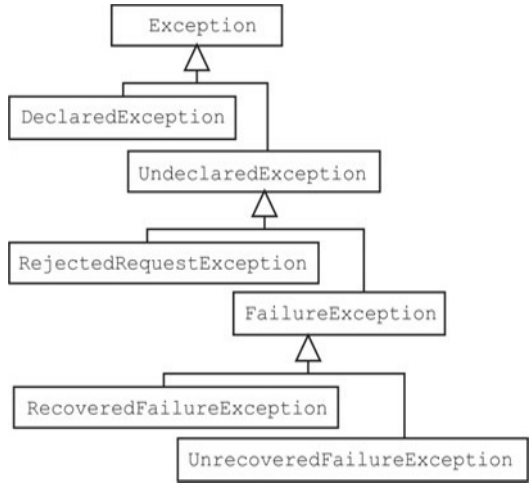

Figure 3. Abstract exception type hierarchy

described applies to both situations: the adaptation of existing components and the development of new components.

When a component is built from scratch, its implementation is under control of the software developer. Figure 4 depicts the proposed internal structure of a component with one provided interface and one required interface. The employed notation is UML 2.0[25]. The implementation classes implement the operations specified by the component's provided interface. Furthermore, these classes may have dependencies that are explicitly represented by means of the required interface of the component. In the proposed strategy, implementation classes are responsible for: (i) detecting exceptional conditions anticipated by the specification of the component and signalling exceptions of types declared in the provided interface of the component; (ii) signalling internal exceptions related to other exceptional conditions which are specific to the implementation of the component; and (iii) executing clean-up actions, when necessary. The types of exceptions raised by the implementation classes should be subtypes of `DeclaredException`

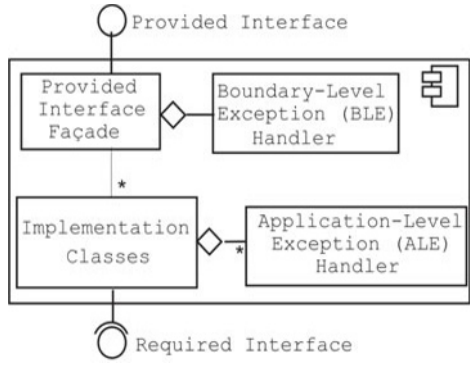

Figure 4. Internal structure of a component built from scratch.

and `UndeclaredException`, depending on whether the exceptional condition that was anticipated or unanticipated.

A façade class[12] is associated to a provided interface and defines an access point to its implementation. When necessary, façade classes could be also responsible for serializing incoming requests, in order to transform the component in a damage confinement region[1]. A façade class may also detect the violation of pre- and post-conditions for operations defined by its corresponding provided interface.

The intra-component strategy is based on application-level exception (ALE) handlers and boundary-level exception (BLE) handlers. ALE handlers are associated to implementation classes. They are responsible for handling three kinds of exceptions: (i) external exceptions of types declared in the required interfaces of the components; (ii) internal exceptions signalled by implementation classes; and (iii) internal exceptions signalled by the underlying infrastructure.

BLE handlers are responsible for dealing with exceptions that reach façade classes. Exceptions of types declared in the provided interface of the component, which are anticipated by its specification, are simply propagated by BLE handlers. These handlers also propagate exceptions of type `RejectedRequestException`, which signal an error in the request issued by the client. For other exception types, backward error recovery may be performed, in case it is available. If the component is left in a consistent state, an exception of type `RecoveredFailure Exception` is signalled, indicating that the state of the component is consistent. Otherwise, an exception of type `UnrecoveredFailureException` is signalled.

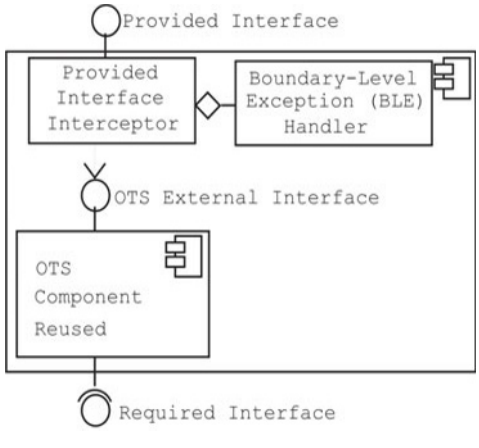

Figure 5. Internal structure of a wrapped OTS component.

When an existing OTS component is reused, a wrapper should be created in order to avoid architectural mismatches (Section 3.3). In this case, the abstract architectural component is instantiated by a composite component wrapping the OTS component, as depicted in Figure 5. The OTS component may include its own exception handlers. All responsibilities that are associated to façade classes and BLE handlers in a component built from scratch, are associated, respectively, to provided interface interceptors and BLE handlers of the composite component. Moreover, a provided interface interceptor is responsible for adapting the OTS external interface to the provided interface specified for the abstract architectural component. Provided interface interceptors, together with the BLE handlers, are also responsible for mapping exceptions raised by the component's implementation to the abstract exception type hierarchy. The main responsibility of the OTS component is the implementation of its external interface.

4.3 Inter-Component Exception Handling Strategy

The inter-component strategy is applied during the assembly workflow of a CBD process (Section 3.3). This strategy deals with the integration of pre-existing components into a new configuration. It is based on connector-level exception (CLE) handlers that are associated to architectural connectors in a specific software configuration. Figure 6 shows the internal structure of an architectural connector with a CLE handler and how it connects a client component to a server component.

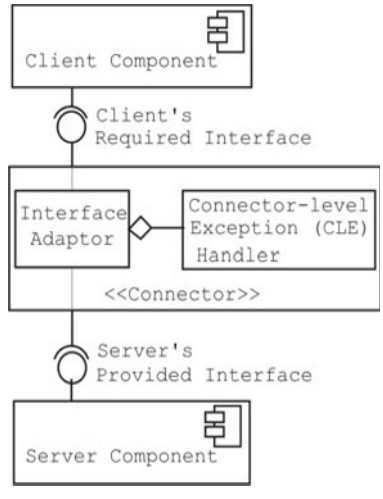

Figure 6. Connector-level exception handler.

CLE handlers are responsible for: (i) providing error recovery and masking at the architectural level exceptions that cannot be handled within the context of a specific component; and (ii) resolving failure semantics mismatches between server components and their clients, for instance, when a server component signals an exception that is not expected by its client. A CLE handler should be capable of dealing with all exceptions signalled by server components. A possible scenario of exception handling at the architectural level is a configuration that includes two or more redundant server components. In this scenario, a fault-tolerant

client/server connector could be used as a mediator between client components and the redundant servers. Upon receipt of an exception from a server component, the associated CLE handler can try to mask the exception by re-invoking the same service on an alternate (replicated or diversely designed[1]) server component. Moreover, if a server component successively fails, generating too many exceptions, the connector may choose to isolate it and forward all subsequent requests to an alternate server component[7,27].

CLE handlers are also responsible for translating the types of unmasked exceptions from the domain of the server component to the domain of the client component, before propagating them. Exceptions that require no further adaptation are automatically propagated. When automatic propagation is not possible, CLE handler can create a new exception that encapsulates the unmasked exception raised by the server component. The type of the propagated exception should be: (i) one of the exception types declared for the operation requested by the client component, as defined by its required interface, or (ii) a subtype of `UndeclaredException`. (Figure 3).

Table 1 provides guidelines for exception translation followed by architectural connectors. These guidelines are based on the configuration depicted in Figure 6 and on the premise that the `Server` component signalled an exception `E1` in response of a service request received from the `Client` component.

CLE handlers are the best candidates for coordinating the exceptional behaviour specified for the integrated system. This way, the implementation of the exceptional behaviour of the integrated system is less dependent of any particular version of a component's implementation. Moreover, connectors are developed during the assembly workflow, when knowledge about the integrated system's requirements, the exceptional behaviour specified for its components, and the way they should interact, is available. Being so, CLE handlers can take reasonable recovery actions based on the abstract types of the undeclared exceptions they may receive (Section 4.1). As these recovery actions are system-dependent, this separation of concerns also improves component reuse.

4.4 A Method for Our Exception Handling Strategy

This section describes a basic method for applying our exception handling strategy. This method was devised as an extension of a typical component-based process and is based on our previous experience in the use of exception handling for building fault-tolerant component-based systems[17,26,29].

Figure 7 shows the main artifacts added by our method and how they integrate in the development process shown in Figure 2. The method starts with the specification of the failure hypotheses and the exceptional behaviour expected for the system. During the requirements workflow, use case descriptions are analysed, in order to extract exceptional scenarios. Next, during the component specification workflow, these scenarios are used to specify the exceptional conditions that should be anticipated by

Exception E1 signalled by the server component	Type of exception propagated to the client component
`E1` is declared in both the required interface of the `Client` component and the provided interface of the `Server` component	`E1` (it may be automatically propagated)
`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	`E2` (e.g. `E1` and `E2` have a common ancestor, `E3`, or `E2` is a super type of `E1`)
`E1` is declared in the provided interface of the Server component and can not be translated according to the two rules stated above	A subtype of `RejectedRequestException`, `UnrecoveredFailureException`, or `RecoveredFailureException`, according to the failure semantics associated to `E1` (Section 4.1)
`E1` is a subtype of `UndeclaredException`	`E1` (it may be automatically propagated)
`E1` is not declared in the provided interface of the `Server` component and is not a subtype of `UndeclaredException`	A subtype of UnrecoveredFailureException

Table 1: Guidelines for exception translation by a connector.

the system and the types of the exceptions that might be signalled. Based on this, the various kinds of handlers (ALE, BLE and CLE) are specified, assigning to them responsibilities for dealing with exception types defined in the previous step. Finally, exceptions types and handlers are implemented and incorporated in the system, during the provisioning and assembly workflows.

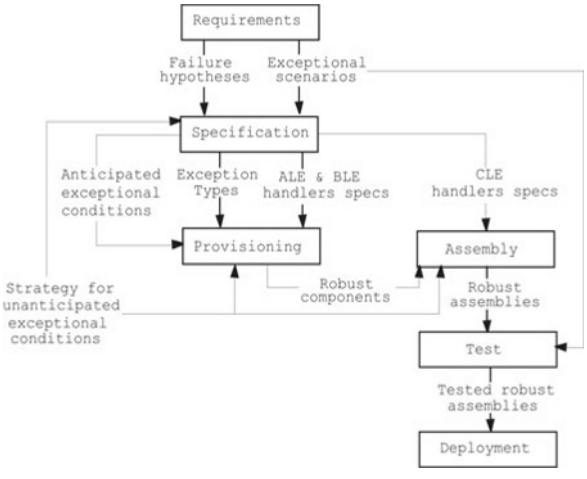

Figure 7. A component-based process extended with our exception handling strategy.

The activities of the proposed method are described next with more details.

Activity 1: Specification of the failure hypotheses for the design of the system's exceptional behaviour. This includes the specification of exceptional conditions to be detected and the exception types that will signal these conditions. Although these two activities are not subject of the present work, they are essential for the development process. Activity 1 should be performed during the requirements workflow.

Activity 2: Design of the exceptional behaviour, allocating responsibilities to the various architectural elements and their exception handlers (ALE, BLE or CLE) and covering the failure hypothesis defined in Activity 1. Furthermore, generic handlers that deal with unanticipated exceptional conditions may be defined, for instance, to trigger backward error recovery in case an undeclared exception is signalled. This activity should be performed during the specification workflow.

Activity 3: Implementation of 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. This activity is performed during the provisioning workflow.

Activity 4: Implementation of ALE and BLE handlers. In Java and C#, this activity is performed by using `try-catch` blocks. We suggest that the actual handlers be implemented as methods in separate classes responsible exclusively for exception handling. In this manner, normal and exceptional behaviour are more explicitly decoupled and exception handlers can be reused. This activity is performed during the component provisioning workflow.

Activity 5: Implementation of CLE handlers. This activity is performed during the assembly workflow.

5 Case Study

In this section, we describe a case study that has been conducted to assess the feasibility and benefits obtained from applying our approach to part of a real system. The main goal of this case study was to analyse the impact of the proposed approach when applied to an existing system, in terms of both separation of normal and exceptional activities and reuse of the implementation of existing components. The target system, called Telestrada, is a large traveller information system being developed for a Brazilian national highway administrator. It comprises five subsystems: Central Database Subsystem, GIS (Geographic Information System) Subsystem, Call-Centre Operations Subsystem, Roadside Operations Subsystem, and Complaint Management Subsystem.

The case study consisted in applying our exception handling strategy presented in Section 4 to the Complaint Management Subsystem (CMS), in order to model its exceptional behaviour. This subsystem is a web-based application implemented in Java using the COSMOS model (Section 3.2). The implementation of the CMS comprises 12175 lines of source code (1598 automatically generated), as measured by the Unix `wc` (word count) command, and more than 300 classes. It is based on popular technologies, such as Enterprise Java Beans, Java Server Pages and Servlets, and the Struts framework.

The case study covered two iterations of the implementation of the CMS. During the first iteration, it was produced an initial implementation of the CMS in which exception handling was introduced in an *ad hoc* manner. The development of the proposed approach occurred after the conclusion of this initial implementation. Hence, the first iteration was not influenced by the proposed approach. During the second iteration, our approach was applied to obtain a robust implementation with a structured exceptional behaviour of the CMS. Another developer that was familiar with the proposed approach but had no previous contact with the CMS conducted this second iteration. Hence, the conditions under which the second iteration was conducted were similar to those of a real software development effort.

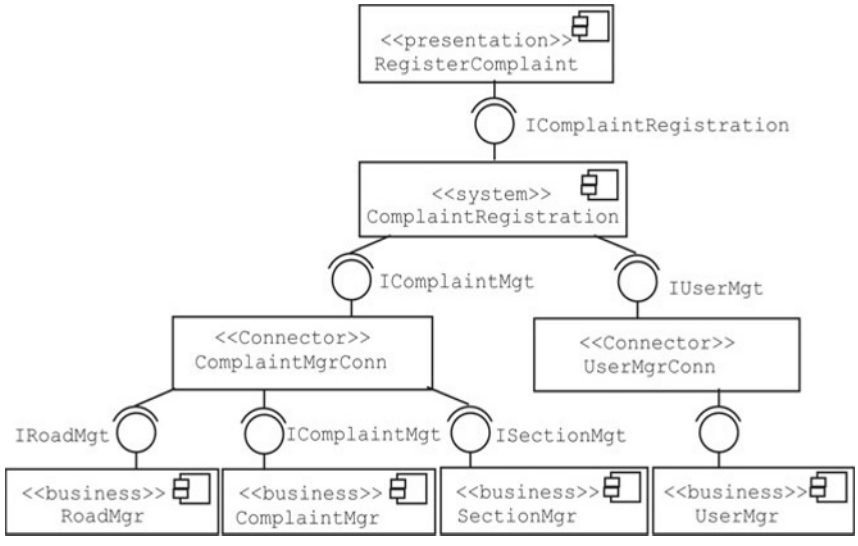

Figure 8. Partial layered architecture of the Complaint Management Subsystem (CMS)

5.1 System Description

The CMS manages the complaints of the users about a section of a road. Information about roads and their sections are stored in a centralized database by the GIS Subsystem. Users register complaints through a form at the website of the project, selecting the desired road section. In this process, the system presents any information about existing complaints for the selected road section. If a complaint is successfully registered, it is recorded in the database and the user receives a complaint id that can later be used to inquire about the complaint status. If the complaint cannot be registered by the system, an error message is presented to the user.

Part of the layered architecture of the CMS is presented in Figure 8.

The component's stereotype indicates the architectural layer to which it belongs. The presentation layer component `RegisterComplaint` uses the `IComplaintRegistration` interface, provided by the `ComplaintRegistration` component of the system layer. The `ComplaintRegistration` component, on its turn, requires services provided by the `ComplaintMgr`, `RoadMgr`, `SectionMgr` and `UserMgr` components of the business layer. The `ComplaintMgrConn` and `UserMgrConn` connectors mediate the interaction between the `ComplaintRegistration` component and those components of the business layer. Components of the business layer use a database management system to store and retrieve persistent information.

The case study was conducted by following the systematic approach described in Section 4.4. For clarity's sake, in this section we focus on the Register Complaint use case.

Use Case: Register Complaint
Main Success Scenario:
1. User selects the "Register Complaint" link in the Telestrada homepage.
2. System presents list of roads.
3. User selects the desired road.
4. System presents road sections of the selected road and the registered types of complaints.
5. User selects a section and a complaint type.
6. System checks if there is a default answer for the selected road section and complaint type.
7. System presents default answer, in case it exists, together with a form where the user may enter a complaint and his personal information (name and e-mail).
8. User fills in the form and selects the option 'Include complaint in database'.
9. System includes the complaint, exhibits the complaint registration number and instructions for the user to check the status of the complaint.

Figure 9. Use case *Register Complaint*.

Exceptional Scenario 7:
Context: step 4 of the main scenario.
Exceptional pre-condition: the system is unable to present to the user the list of road sections corresponding to the selected road
Signal: problems when trying to access the database server.
Handler: prepare an error message explaining to the user that the database is not accessible.
Exceptional post-condition: an error message is presented to the user and the system state is left unmodified.

Figure 10. Exceptional scenario 7.

5.2 Specification of the failure hypotheses

The main success scenario for the Register Complaint use case is shown in Figure 9. For each exceptional condition in this scenario, an exceptional scenario was defined. In total, 11 exceptional scenarios were defined for the Register Complaint use case. The exceptional scenario 7 is described in Figure 10.

5.3 Design of the exceptional behaviour

The UML Components process[6] suggests that, for each step in the main scenario of a use case, an operation in a system layer interface should be defined. We performed this activity during the first iteration of the case study. In the second iteration, the operations were specified in terms of pre- and post-conditions, and the exceptions they might signal.

For instance, step 4 of Register Complaint use case was mapped to the `listRoadSections` operation of the `IComplaintRegistration` interface (Figure 8). The specification of this operation, as well as the exceptions that it may signal, are described in Figure 11.

Operation: `IComplaintRegistration::listRoadSections`
Pre-condition: some section of the selected road must exist.
Post-condition: the list of sections for the selected road is returned.
Exception associated to pre-condition: `SectionsDoNotExistException`
Exception associated to post-condition: `ListSectionException`

Figure 11. Operation listRoadSection

After specifying the exceptional failure hypotheses for all the exceptional scenarios, we defined the exceptional behaviour of the system for each of them. For the `listRoadSections` operation, the exceptional behaviour consists of preparing an error message and displaying it to the user. In fact, this exceptional behaviour is adopted by almost all the operations in the system. In some cases, rollback is also performed, in order to guarantee the consistence of the system's state.

5.4 Implementation of the exceptional behaviour

For most architectural components of the CMS the initial implementation was modified to adhere to the internal structure shown in Figure 4 (Section 4.2). However, the initial implementation of the `ComplaintMgr` component was reused as an OTS component. Hence, the internal structure of the new `ComplaintMgr` component resembles Figure 5. In our implementation, ALE, BLE, and CLE handlers were developed as classes where methods correspond to the actual handlers.

Façade classes of components were modified to introduce explicit checks for pre- and post-conditions. In the initial implementation, pre- and post-conditions of operations were checked at the presentation layer, or not at all. In the robust implementation, façade classes raise an exception of type `RejectedRequestException`, if a service request violates a pre-condition. When a response violates a post-condition, the façade class invokes the appropriate BLE handler. In most cases, the BLE handler signals an exception of a subtype of `RecoveredFailureException` or `UnrecoveredFailureException` (Section 4.1).

The use of BLE handlers guarantees that components always produce meaningful responses, when errors occur. For instance, during the execution of the `listRoadSections` operation, if the implementation classes of the `SectionMgr` component signal an exception of type `SectionDatabaseQueryException`, this exception is automatically propagated, since it denotes an anticipated exceptional condition. However, if an exception of type `NullPointerException` reaches the component boundary, it is treated as signalling an unanticipated exceptional condition. Hence, the handler encapsulates this exception as an instance of `RecoveredFailureException` and raises it. The component's state is guaranteed to be consistent, since it is not modified by the implementation of `listRoadSections`. If support for backward error recovery is available and the failed operation modifies the system's state, the BLE handlers may try to restore it to a previous state free of errors.

The code snippet in Figure 12 illustrates how the provided interface interceptor of the robust implementation of the `ComplaintMgr` interceptor works.

The `IManager` interface defines methods for managing the dependencies of the component. Each component in the system provides its own implementation. In the example, the interceptor uses an object of type `IManager` to obtain a reference to the original `ComplaintMgr` component.

The class `ComplaintMgtInterceptor_Exceptional` implements the exceptional behaviour of the component. This class implements a polymorphic `handle` method responsible for handling the exceptions that may be raised by the operations of the original component. In this example, handling consists of transforming instances of `SQLException` in instances of `ComplaintRegistrationException`.

```
1 package business.complaintMgr.impl;
2 // imports all the required types
3 public class ComplaintMgrInterceptor  implements IComplaintMgt {
4  private IManager manager;
5  public ComplaintMgtInterceptor(IManager manager){
6    this.manager = manager;
7  }
8  public String registerComplaint(...) throws ComplaintRegistrationException
9 {
10    IComplaintMgt iComplaintMgt = (IComplaintMgt)manager.
11    getRequiredInterface("business.complaintMgr.spec.prov.IComplaintMgt");
12    String complaintId =  new String();
13    try {
14      complaintId = iComplaintMgt.registerComplaint(...);
15    } catch (SQLException e) {
16      ComplaintMgtInterceptor_Exceptional handler =
17        new ComplaintMgtInterceptor_Exceptional();
18      handler.handle(e);
19    }
20    return complaintId;
21  }
22  (...)
23 }
```

Figure 12. Implementation of the `registerComplaint` method of class `ComplaintMgrInterceptor`

We have also modified the initial implementation of the connectors `ComplaintMgrConn` and `UserMgrConn`. In both cases, very small modifications were made to their interface adaptor classes (Section 5). For each of them, a new class responsible for the exceptional behaviour was implemented, much like the `ComplaintMgrInterceptor_Exceptional` in the example above. These CLE handlers were responsible for dealing with unanticipated exceptional conditions. If a server component raises an unexpected exception, the connector is responsible for transforming it into some declared exception defined by the exceptional contract of the client component or, if this is not possible, an appropriate subclass of `FailureException`.

5.5 Discussion

The most important benefit of applying our approach to the CMS was enhanced system structure. The implementation of the system bore a greater resemblance to its design and the code responsible for the exceptional behaviour was more clearly separated. The main consequence of this fact is decreased complexity. Moreover, exceptions are confined to where they are semantically meaningful. For instance, in our case study, the system layer did not have to handle any exception directly related to the database. Hence, components are more reusable, because they only have to deal with exceptions that are directly related to their conceptual domains.

Component-based systems built according to our proposed approach are also more robust, for two main reasons:

1. The specific exception handlers share responsibilities for exception handling at different levels of semantics abstraction. The most concrete and implementation-dependent level is assigned to ALE handlers, while the most abstract and system-dependent level is assigned to CLE handlers. Thus, specific exception handling strategies can be employed at each semantic level, preserving the system's independence of its component's implementations. This improves the substitutability of the system's components and, hence, also improves its robustness.

2. Our exception handling strategy includes concrete guidelines about how a component should react to unanticipated exceptional conditions. This kind of design decision is not left open to the component developer. This avoids the situation where, in absence of a specification, the developers adopt bad practices such as: swallowing an exception, including an empty `catch` block, and propagating an exception that is meaningless to the component's client.

CLE handlers guarantee that exceptions a component receives are compatible with the abstract exception

type hierarchy, as discussed in Section 5.2. Hence, there is always some semantic information regarding the error that can be taken into account by handlers. Furthermore, the exceptions in the abstract exception type hierarchy provide some information regarding the state in which the component that signalled the exception was left. This is important for error diagnosis and recovery.

The intra-component strategy for reusing OTS components proved to be useful. To reuse the initial implementation of the `ComplaintMgr` component, no modifications were required to other elements of the system. All we had to do was to deploy the robust implementation (reused `ComplaintMgr` plus provided interface interceptor and BLE handlers) as if it were the initial implementation. This is an important benefit of applying the proposed approach.

The implementation overhead of applying our strategy in an existing system is not negligible. In our case study, the robust implementation of the CMS had 9.54% more lines of code than the initial implementation (9747 loc of the original version against 10677 loc of the new one, ignoring automatically generated code). Although the robust implementation is larger, it is also better structured.

Our strategy also imposed a development time overhead, due to specification and implementation activities. The time it took for the developer to perform the three activities described in Section 4.4 throughout requirements, specification, provisioning, and assembly workflows accounted for more than 30% of the time required for the development of the initial implementation.

6 Conclusions

The main contribution of this paper is a general strategy for exception handling in component-based systems, addressing the problem of how to 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[8,22] and combined them in a set of guidelines for structuring exception handling at both architectural and implementation levels.

An initial assessment of the approach described in this paper has been presented elsewhere[15]. Our present work improves this initial assessment adding a new type of exception handler. Pagano[26] describes an extended version of the case study presented in Section 5. Guerra[17] presents a case study describing the application of the proposed exception handling strategy to a real-world banking application.

Although the workflow described in Section 4.4 may be used in isolation, it is more effective if fully integrated with a CBD process. In this manner, it can be refined and the specification of the exceptional behaviour of a system can be taken into account since early stages of development. We are currently extending the UML components process[2] with the method described in Section 4.4. This effort builds upon previous work on the definition of a CBD process that takes the exceptional behaviour of a system into consideration[29].

Our most immediate future work consists of developing 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[37].

Other important issues to be addressed in future works are: (i) to measure quantitatively the impact of the proposed approach in the reliability of the final system; and (ii) to investigate how the proposed approach can be extended to include guidelines for structuring concurrent exception handling. For the reliability analysis, our intent is to apply fault-injection techniques on both implementations of the Complaint Management Subsystem to obtain statistical data about the frequency of failures before and after the application of the proposed approach. The structuring of concurrent exception handling, at the architectural level, is currently being addressed by our research.

Furthermore, we intend to evaluate the applicability of aspect-oriented programming[11] techniques to increase separation of concerns in two complementary levels. First, to specify architectural level exception handlers. In this case, aspects would complement existing architecture description languages, instead of programming languages. The result of weaving such aspects would be an extended architecture description that expresses certain properties regarding dependability. Second, to help in decoupling the implementation of the normal and exceptional behaviours of systems built according to the proposed guidelines. These are both ongoing works that are described in more detail elsewhere[4,5].

Acknowledgements

F. Castor Filho is supported by FAPESP/Brazil, grant number 02/13996-2. C. M. F. Rubira is partially supported by CNPq/Brazil, grant number 351592/97-0. We would like to thank Rodrigo Tomita for reading a draft of the paper and providing several interesting comments. We are also grateful to the anonymous reviewers for their valuable remarks that contributed to many improvements in this final version of the paper.

References

[1] T. Anderson and P A. Lee. Fault Tolerance: Principles and Practice. Springer-Verlag, Wine, Austria, 2nd Edition, 1990.

[2] P. H. S. Brito, F. C. Filho and C. M. F. Rubira. A method for modeling exceptions in component-based development (in portuguese). In Proc. IV Brazilian Workshop on Component-Based Development (WDBC'2004), pp. 29-34, João Pessoa, PB, Brazil, Sep. 2004.

[3] F. Castor Filho, P. A. de C. Guerra and C. M. F. Rubira. An Architectural-level Exception Handling System for Component-based Applications. In: Proc. First Latin-American Symposium on Dependable Computing, LNCS 2847, pp. 321-340, Springer-Verlag, 2003.

[4] F. Castor Filho and C. M. F. Rubira. Implementing Coordinated Error Recovery for Distributed Object-Oriented Systems in AspectJ. Journal of Universal Computer Science, 10(7):843-858, Jul. 2004.

[5] F. Castor Filho, P. H. S. Brito, and C. M. F. Rubira. A Framework for Analyzing Exception Flow in Software Architectures. Submitted to IV ICSE Workshop on Architecting Dependable Systems (WADS'2005).

[6] J. Cheesman and J. Daniels. UML Components: A Simple Process for Specifying Component-Based Software. Addison-Wesley, Reading, MA., USA, Oct. 2000.

[7] J. E. Cook and J. A. Dage. Highly reliable upgrading of components. In Proc. 21st International Conference on Software Engineering (ICSE'1999), pp. 203-212, Los Angeles, CA, ACM Press, May 1999.

[8] F. Cristian. Exception Handling. In: T. Anderson (Ed.) Dependability of Resilient Computers. BSP Professional Books, UK, pp. 68-97, 1989.

[9] G. Doshi. Best practices for exception handling. ONJava Website. November 2003. http://www.oreillynet.com/pub/a/onjava/2003/11/19/exceptions.html

[10] D. D'Souza and A. C. Wills. Objects, Components and Frameworks with UML: The Catalysis Approach. Addison-Wesley, 2nd edition, 1999.

[11] T. Elrad, R. E. Filman and A. Bader. Aspect-oriented programming. Communications of the ACM, 44(10):28-32, 2001.

[12] E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns: Elements of Reusable Software Systems. Addison-Wesley, 1995.

[13] D. Garlan, R. Allen and J. Ockerbloom. Architectural Mismatch: Why Reuse Is So Hard. IEEE Software. 12(6):17-26, 1995.

[14] D. Garlan, R. T. Monroe and D. Wile. Acme: Architectural Description of Component-Based Systems. In: G. T. Leavens and M. Sitamaran (Eds.) Foundations of Component Based Systems, chapter 3, pp. 47-67. Cambridge University Press, Cambridge, UK. 2000.

[15] P. A. de C. Guerra, F. Castor Filho V. A. Pagano and C. M. F. Rubira. Structuring exception handling for dependable component-based software systems. In Proc. 30th Euromicro Conference, Rennes, France, IEEE Computer Society Press. Sep. 2004.

[16] P. A. de C. Guerra, C. M. F. Rubira and R. de Lemos. A Fault-Tolerant Software Architecture for Component-Based Software Systems. In Architecting Dependable Systems. LNCS 2677. Springer-Verlag. 2003.

[17] P. A. de C. Guerra. An Architectural Approach for Fault Tolerance in Component-Based Software Systems (in portuguese). PhD thesis, Universidade Estadual de Campinas, 2004.

[18] V. Issarny and J. P. Banatre. Architecture-Based Exception Handling. In Proc. 34th Annual Hawaii International Conference on System Sciences (HICSS'34). IEEE Computer Society Press, 2001.

[19] A. Kalakech et al. Benchmarcing operating system dependability: Windows 2000 as a case study. In Proc. 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'2004), pp. 261-270, Papeete, Tahiti, IEEE Computer Society Press, Mar. 2004.

[20] M. D. McIlroy. Mass-Produced Software Components. In P. Naur and B. Randell (Eds) Software Engineering. Petrocelli/Charter, Brussels, Belgium, pp. 88-98. 1976.

[21] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In Proc. 6th Joint ACM/Sigsoft Symposium on Foundations of Software Engineering and European Software Engineering Conference (FSE/ESEC'97), Sep. 1997.

[22] B. Meyer. Object-Oriented Software Construction. Prentice-Hall, New Jersey, 1988.

[23] B. Meyer. The grand challenge of trusted components. In Proc. 25th International Conference on Software Engineering, pp. 660-667. IEEE Computer Society Press, May 2003.

[24] Microsoft Corporation. Microsoft .Net Information. Available at http://www.microsoft.com/net/

[25] Object Management Group. Unified Modeling Language: Superstructure, version 2.0. Jul. 2003.

[26] V. A. Pagano. An architectural approach based on exception handling for the design of component-based software systems (in portuguese). Master's thesis, Universidade Estadual de Campinas, 2004.

[27] M. Rakic and N. Medvidovic. Increasing the confidence in off-the-shelf components: A software connector-based approach. In Proc. 2001 Symposium on Software Reusability, pp. 11-18. ACM/SIGSOFT, May 2001.

[28] D. Reimer and H. Srinivasan. Analyzing exception usage in large java applications. In Proc. ECOOP'2003 -Workshop on Exception Handling for Object-Oriented Systems, pp. 10-19, Darmstadt, Germany, Jul. 2003.

[29] C. M. F. Rubira, R. de Lemos, G. Ferreira and F. Castor Filho. Exception handling in the development of dependable component-based systems. Software - Practice and Experience, 2005.

[30] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Morgan Kaufmann Publishers, 1996.

[31] S. Shenoy. Best practices in EJB exception handling. In IBM developerWorks website. Available at http://www-106.ibm.com/developerworks/library/j-ejbexcept. 2002.

[32] J. Siedersleben. Errors and exceptions - rights and responsibilities. In Proc. ECOOP'2003 -Workshop on Exception Handling for Object-Oriented Systems, pp. 2-9, Darmstadt, Germany, Jul. 2003.

[33] M. Silva Jr., P. A. de C. Guerra and C. M. F. Rubira. A Java Component Model for Evolving Software Systems. In Proc. 18th IEEE International Symposium on Automated Software Engineering, pp. 327-330, Oct. 2003.

[34] Sun Microsystems. Enterprise javabeans specification v2.1 - proposed final draft, 2002. Available at http://java.sun.com/products/ejb/

[35] Sun Microsystem. Java 2 Platform, Enterprise Edition (J2EE). Available at http://java.sun.com/j2ee/index.jsp

[36] C. Szyperski. Component Software: Beyond Object-Oriented Programming. ACM Press and Addison-Wesley, New York, NY, second edition, November 2002.

[37] R. T. Tomita, F. Castor Filho, P. A. de C. Guerra and C. M. F. Rubira. Bellatrix: An environment with architectural support for component-based development (in portuguese). In Proc. IV Brazilian Workshop on Component-Based Development (WDBC'2004), pp. 43-48, João Pessoa, PB, Brazil, Sep. 2004.

[38] G. Veccellio and W. M. Thomas. Issues in the assurance of component-based software. In Proc. 2000 International Workshop on Component-Based Software, Carnegie Mellon Software Engineering Institute, 2000.

[39] 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 Proc. IEEE 25th Int. Symp. on Fault-Tolerant Computing, pp. 499-508, Pasadena, 1995.

[40] J. Xu, B. Randell, A. Romanovsky, R. J. Stroud, A. F. Zorzo, E. Canver and F. von Henke. Rigorous development of an embedded fault-tolerant system based on coordinated atomic actions. IEEE Transactions on Computers, 51(2):164-179, Feb. 2002.