An Exception Handling Mechanism for Developing Dependable Object-Oriented Software Based on a Meta-Level Approach

Alessandro F. Garcia Delano M. Beder Cecília M. F. Rubira

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

Abstract
The current lack of effective error-handling techniques for developing dependable object-oriented software produces software components which are usually difficult to understand, to change and to maintain in the presence of faults. Ideally such components should incorporate their exceptional activity in a structured and transparent manner so that the normal code would not be amalgamated to the normal code. In this context, we propose the design and implementation of an object-oriented exception handling mechanism based on a meta-level approach. This approach is based on a computational reflection mechanism which encourages modular descriptions of software systems by introducing a new dimension of modularity — the separation of the base-level computation from the meta-level computation.

The goal of our work is twofold: (i) to define an exception handling model which supports a clear and transparent separation of the normal activity of a component from its exceptional activity, and (ii) to provide a meta-level architecture which implements such mechanism. Our exception handling model consists of the following characteristics: (i) exceptions are represented as data objects [7, 11]; (ii) exception handlers are represented as ordinary methods; (iii) the creation of exceptional class hierarchies which implement exception handlers, that are orthogonal to the application’s normal class hierarchies; (iv) the attachment of handlers can occur at different levels: (1) methods, (2) individual objects or groups of objects, (3) classes, and (4) exceptions; and (v) support for concurrency and coordinated error recovery.

Our mechanism does not require any special language support and was implemented within the Java programming language without any changes to the language itself by means of a meta-object protocol.

1. Introduction

With software systems growing in size and complexity, the quality and cost of developing and maintaining them are still deep concerns for software developers. Object-oriented component-based engineering is a promising approach in reducing software development cost while increasing productivity, reusability, quality and dependability of software systems and their components. However, the development of dependable object-oriented software requires suitable exception detection and handling mechanism to satisfy the system’s dependability requirements.

The current lack of effective error-handling techniques for developing dependable object-oriented software produces software components which are usually difficult to understand, to change and to maintain in the presence of faults. Ideally such components should incorporate their abnormal behavior (i.e., their exceptional activity) in a structured and transparent manner so the abnormal code would not be amalgamated to the normal code. In this context, we propose the design and implementation of an object-oriented exception handling mechanism based on a meta-level approach. This approach is based on a computational reflection mechanism which encourages modular descriptions of software systems by introducing a new dimension of modularity — the separation of the base-level computation from the meta-level computation.

The goal of our work is twofold: (i) to define an exception handling model which supports a clear and transparent separation of the normal activity of a component from its exceptional activity, and (ii) to provide a meta-level architecture which implements an exception handling mechanism. Our exception handling model consists of the following characteristics: (i) exceptions are represented as data objects [7, 11]; (ii) exception handlers are represented as ordinary methods; (iii) the creation of exceptional class hierarchies which implement exception handlers, that are orthogonal to the application’s normal class hierarchies; (iv) the attachment of handlers can occur at different levels: (1) methods, (2) individual objects or groups of objects, (3) classes, and (4) exceptions; and (v) support for concurrency and coordinated error recovery.

Our mechanism does not require any special language support and was implemented within the Java programming language without any changes to the language itself by means of a meta-object protocol called Guaraná [18]. The remainder of this text is organized as follows. Section 2 defines the terminology adopted in this work related to exception handling and fault tolerance. Section 3 discusses some important design issues related to exception handling mechanisms in object-oriented languages and concurrent systems. Section 4 presents the concepts of computational reflection and meta-level architectures. Section 5 presents our exception handling
model. Section 6 describes an example of use of the proposed mechanism. Section 7 describes our meta-level architecture for exception handling. Section 8 shortly explains a comparison with related work. Finally, Section 9 summarizes the conclusions of this work and suggests directions for future work.

2. Exception handling and fault tolerance

Following the terminology adopted by Lee and Anderson [12], a system consists of a set of components that interact under the control of a design. A fault in a component may cause an error in the internal state of the system which eventually leads to the failure of the system. Two techniques are available for eliminating the errors from the system’s state: (i) forward error recovery and (ii) backward error recovery. The first technique attempts to return the system to an error-free state by applying corrections to the damaged state. The second technique attempts to restore a previous state which is presumed to be free from errors. Although traditionally exceptions and exception handling constitute a common mechanism applied to the provision of forward error recovery, they may provide support to combine forward and backward error recovery schemes [1]. Therefore, the notions of exceptions and exception handling can be used to establish a framework for achieving fault tolerance.

Software components receive service requests and produce responses. If a component cannot satisfy a service request, it returns an exception. So the responses from a component can be separated into two distinct categories, namely normal and exceptional responses. To create a clear framework, the activity of a component can be divided in two parts: normal activity and abnormal (or exceptional) activity (Figure 1). The normal activity implements the component’s normal services while the exceptional activity provides measures for tolerating faults that cause such exceptions. Thus, the normal activity of the system is clearly distinguished from its exceptional activity.

Exceptions can be classified into three different categories: (i) interface exceptions which are signaled in response to a request which did not conform to the component’s interface; (ii) failure exceptions which are signaled if a component determines that for some reason it can not provide its specified service; (iii) internal exceptions which are exceptions raised by the component in order to invoke its own internal exceptional activity. Note that an exception is raised within the component, but signaled between components. Whenever an exception is raised in a component that does not have a handler for it, the exception is signaled to the component (caller) that dynamically invoked the first one. If no handler is defined for an exception within the caller, the exception is propagated to higher-level components. To each level of the system, a component called idealized fault-tolerant component [12] handles exceptions produced by lower-level components or propagates them for higher-level components.

Figure 1. Idealized fault-tolerant component [12]

Because faults are expected to occur rarely during the component’s normal activity, programmers usually refer to them as exceptions. Exception handling mechanisms (or merely exception mechanisms) are often provided in programming languages and allow software developers to define exceptional conditions and to structure the exceptional activity of software components. When an exception is detected in a component, such a mechanism is responsible for changing the normal control flow of a computation within a component to the exceptional control flow. Therefore, raising an exception results in the interruption of the component’s normal activity, followed by the search for an exception handler (or simply handler) to deal with the raised exception. The set of handlers of a component constitutes its exceptional activity part. For any exception mechanism, handling contexts associate exceptions and handlers. Handling contexts are defined as regions in which the same exceptions are treated in the same way. Each context should have a set of associated handlers, one of which is called when the corresponding exception is raised.

3. The design of exception mechanisms

There are some important issues that should be considered during the design of an exception mechanism. In this Section we discuss them in turn.

**Exception representation.** Exceptions can be represented as (i) names, (ii) data objects, or (iii) full objects. Representing exceptions as names is a classical approach adopted by several object-oriented programming languages, such as Eiffel [15]. In the second category, exceptions are classes and an instance of an exception
class is created every time that an exception is raised. The main task of raising an exception is to pass an exception object as a parameter to the corresponding handler. C++ [11] and Java [7] adopt this approach. In the third category, exceptions are also organized hierarchically as classes and the task of raising an exception is to create an instance of the related exception class and then call it with a raise() or throw() operation. In this case, the exception is a standard object that receives messages. The exception handling system implemented in Lore [5] applies this design solution.

**Placement of Handlers.** Handling contexts can be defined as: (i) a statement or a block, (ii) a method, (iii) an object, (iv) a class, or (v) an exception. Statement (or block) handlers are attached to a statement (or a block of instructions), allowing context-dependent responses to an exception. Method handlers are associated to methods; when an exception is raised within the method’s code, the method handler bound to this exception is executed. Object handlers are associated with objects variables in their declaration; that is, each instance has its own set of handlers. Class handlers are attached to classes, allowing the software developers to define a common exceptional behavior for a class in exceptional situations. When handlers are associated with exceptions themselves, they are always invoked if no more specific handler can be found. They are the most general handlers and must be valid in any case, independent of any execution context and object state.

**Exception Propagation.** The exception propagation to higher-level components can be performed in two ways: automatic or explicit. In the first case, if no handler is found for the exception within the caller, the exception is propagated automatically to higher-level components until a handler can be found; that is, an exception can be handled by components other than its immediate caller. In the second case, the handling of signaled exceptions is limited to the immediate caller.

**Continuation of the Control Flow.** When the handler terminates normally, the related exception is said to be handled. Then the system can return to its normal activity; however, there is an issue concerning whether the internal activity of the component that raised the exception can be resumed or not. In the termination model, execution will continue from the point at which the exception was handled. Conceptually, this means that the component activity which raised the exception cannot be resumed. In the resumption model, the handler has the capability to resume the internal activity of the component after the point at which the exception was raised.

**Support to Coordinated Recovery.** Very few object-oriented languages support concurrent exception handling, e.g. to call handlers in several concurrent objects when an exception has been raised in one of them. For instance, Arche language [10] allows user-defined resolution of multiple exception amongst a group of objects that belong to different implementations of a given type; however, this approach is not generally applicable to the coordinated recovery of multiple interacting objects of different types.

### 3.1. Exception handling and the object model

Even though many object-oriented languages provide exception-handling facilities, a few of them provide an exception mechanism that is really integrated with the object model. Classical design issues of exception mechanisms should be re-visited in the light of object-orientation so that exception handling itself could benefit from object-oriented features. For instance, we advocate that the object-oriented design of an exception mechanism should support exception representation as data or full objects. The majority of the object-oriented languages have adopted the exception representation as names. Although it is the classical approach, it does not have a close integration between the object-oriented language and the exception mechanism.

It is also an important issue how to relate exception raising to interface checking [16]. In object-oriented programming, each operation (or method) in a type (or class) description is defined by a signature, which specifies the name of the method and the types of its parameters. Method’s signature should include the exceptional responses that an object may return. For example, Java [7] allows to declare the exceptions that a method may signal in its signature with a throws clause. Nevertheless, when the type specification includes this declaration, new problems arise as to inheritance and subtyping rules. In the subtyping/conformance relationship, a derived class is designed by including the specification of the base class as a subset of its specification. Note that the modification of a method’s signature is not allowed when redefining a method. This implies that the redefinition of operations by derived classes should inherit all exceptions specified by the base class.

Furthermore, for usability and program readability, it is necessary to permit considerable flexibility in the placement of handlers. Thus, an object-oriented exception handling approach should provide different levels of handler attachment. When an exception is related to a method, a handler for this exception may be locally associated with the method. Alternatively, handlers can be associated with a class, which can be applied to all methods of that class. It is also possible to attach handlers to objects themselves.
3.2. Exception handling in concurrent OO systems

In an object-oriented software system, there may be a number of processes (threads) running concurrently. There are different ways of dealing with concurrency in object-oriented systems. In this work, we define a clear distinction between objects and threads: threads are agents of computation that execute operations on objects (which are the subjects of computation). In this sense, concurrent threads can be classified in three categories [12]: (i) independent, (ii) competing, or (iii) cooperating. They are said to be independent if the sets of objects accessed by each thread are disjoint while they are said to be competing when the sets of objects accessed by each thread are not. Threads are said to be cooperating when they are designed collectively and have shared access to common objects that are used directly for communication between the threads.

From the standpoint of fault tolerance, the case of independent threads is trivial; the provision of error recovery to a number of independent threads is identical to the provision of error recovery to a single sequential thread. In the case of competitive threads, the provision of recovery is similar to the first case, but the set of objects accessed by competitive threads should be restored to error-free state as well. In practice, such objects often have their own error recovery scheme. The implementation of an exception mechanism for concurrent systems is an interesting challenge due to cooperative concurrency. The handling of an exception may involve multiple concurrent components when they are cooperating for executing some task. Erroneous information may have been spread directly or indirectly through inter-thread communication. When one of the concurrent threads has raised an exception, error recovery should proceed in a coordinated way by triggering appropriated handlers for the same exception within all the threads [19].

Furthermore, due to nature of concurrent systems, it is possible that various exceptions may be raised concurrently by threads of the system. A structured exception represents the concurrent occurrence of two or more simple exceptions. Exceptions raised concurrently may be the symptoms of a different and more serious fault [19]. In this way, an exception resolution procedure is necessary to select a suitable handler for the exceptions raised concurrently; in this case, such a more generic handler also should be called in all the threads. The work of Campbell and Randell [1] describes a resolution model called exception tree that includes an exception hierarchy imposing a partial order on exceptions of the system. The exceptions that are not listed within the exception tree are categorized as the universal exception. The universal exception is the root of the exception tree. Such a model is used in order to find the exception that represents all the exceptions raised concurrently. So the exception mechanism must activate the handler attached to this more generic exception in every one of the concurrent threads.

4. Reflection and meta-level architectures

Computational reflection [14, 18] is defined as being the ability of observing and manipulating the computational behavior of a system through a process called reification. This technique allows a system to maintain information about itself (meta-information) and use this information to change its behavior. It defines a meta-level architecture which is composed of a base level and a meta-level. A meta-object protocol (MOP) establishes an interface among the base-level and the meta-level components. The MOP provides a high-level interface to the programming language implementation in order to reveal the program information normally hidden by the compiler and/or run-time environment. As a consequence, programmers can develop language extensions and adapt component behavior and even make changes into the system.

Actions that extend the behavior of base-level objects are implemented in the meta-level. Reflection can be used to intercept and modify the effects of operations of the object model. For the purpose of illustration, suppose that for each base-level object o exists a corresponding meta-object mo that represents the behavioral and structural aspects of o. As illustrated in Figure 2, if an object x sends a message service to an object o, the meta-object mo intercepts the message service, reifies the base-level computation and takes over execution; later mo returns (reflects) the response to x. From the point of view of object x, computational reflection is transparent: x sends a message requesting a service to o, and receives the response with no knowledge that the message was intercepted and redirected to the meta-level.

5. An OO exception handling model

The model that we have defined was primarily designed to facilitate the development of dependable and reusable software components. In this Section we present the main characteristics of our exception handling model and
discuss our design choices for each one of the major design issues described in Section 3.

As discussed in Section 2, a system may be composed of a set of idealized fault-tolerant components. In this work, we assume that software designers structure their applications by creating a set of normal classes which implement the normal activities of the software components, and exceptional classes which implement the abnormal activities (Figure 3). Therefore, exceptional classes implement the abnormal activity of the application and they are associated to the corresponding normal classes. In Figure 3, the methods of the exceptional class ExceptionalSupClient are the handlers for the exceptions that should be treated within methods of the class SupClient. Designers may compose an exceptional class hierarchy that is orthogonal to the normal class hierarchy of the application. The exceptional classes ExceptionalSupClient and ExceptionalClient are organized hierarchically so that the resultant hierarchy is orthogonal to the normal class hierarchy (SupClient and Client). Exceptional class hierarchies allow exceptional subclasses inherit handlers from their superclasses and, consequently, they allow exceptional code reuse.

5.1. Exception representation

In our model, exceptions are represented as data objects. Different types of exceptions are organized hierarchically as classes. The class Exception is the root of this hierarchy. Figure 4 shows this exception class hierarchy which represents the exceptions that may be raised during the execution of application’s methods (E1, E2, E3, E4, E5 and E6). The class GroupException extends the class Exception and allows the definition of exceptions that may be raised by cooperating threads needing coordinated recovery (Section 5.5).

Exceptional responses that may be signaled by a method must be described in its method’s signature by means of a throws clause. Figure 3 shows that the method m3() may signal the exceptions E1, E2, E4, E5 or E6.

Let us remark here that due to the base subtyping relation, a handler defined for an exception E is eligible for any exception, subtype of E. Permitting several exceptions to be named in the same handler avoids code replication when the exceptions are all handled in the same way.

5.2. Placement of handlers

We provide support for multi-level attachment of handlers. Handlers may be associated to: (i) an exception, (ii) a class, (iii) an object, or (iv) a method. Firstly, handlers may be associated to exceptions themselves (default handlers). Default handlers are executed when there is not a more specific handler in application.

Handlers may be also associated to a class. In this case, an exceptional class should be created. In Figure 3, the ExceptionalSupClient’s methods are class handlers for the exceptions that should be treated within SupClient’s methods. In the same way, the ExceptionalServer’s methods are class handlers for the exceptions that should be handled within Server’s methods. Nevertheless, the class handlers for the exceptions that should be treated within Client’s methods can be ExceptionalClient’s methods or methods that are inherited from subclasses of the class ExceptionalClient. Therefore, the handler for the exception E5 (E5Handler()) is inherited from the ExceptionalSupClient.

In addition, object handlers may be also defined. To implement handlers associated to individual objects, a new exceptional class must be created. This new class contains methods that implement the object handlers for the exceptions that should be treated in any method of the object. For instance, the object client1, instance of the class Client, may be associated to handlers that are distinct from the handlers that are associated to the object client2 that is also an instance of the class Client (Figure 5). The Exceptional_client1’s methods are object handlers for the exceptions that should be treated within the object client1. Furthermore, it is possible that a single exceptional class be associated to object groups. For example, object handlers associated to client3, instance of the class Client, could be the same handlers associated to client2, i.e., these objects may be associated to a single exceptional class (Exceptional_client2). Thus, client2 and client3 have identical abnormal behavior, while client1 has a different one; although they are instances of the same class. Practical studies [4] have shown that the use of object handlers can produce better structured programs, facilitating its understanding, maintenance and reuse.

![Figure 3. Normal and exceptional class hierarchies](image-url)

![Figure 4. The exception class hierarchy](image-url)
Finally, handlers may be associated to methods. For example, the handler `m2E6Handler()` of the exceptional class `Exceptional_client1` is activated when an exception should be treated in operation `m2()`.

![Diagram]

**Figure 5. Objects and their exceptional classes**

The search of handlers for raised exceptions is defined as follows: (i) if exists an exceptional class attached to the object, the mechanism tries to find method or object handlers associated to the method raising the exception; (ii) if none is found, the system tries to find handlers in exceptional classes or superclasses attached to the normal class of the object; (iii) if none is still found, the exception is then signaled to the caller object and the steps (i) e (ii) are again repeated; (iv) if none is found, the system looks for default handlers attached to the signaled exception itself. Consequently, when `m1()` invokes `m3()`, the internal exception `E3` may be raised. If so does, the exception mechanism activates the local class handler `E3Handler()`. The method `m3()` may signal the exceptions `E1`, `E2`, `E4`, `E5` or `E6`. Supposing `m3()` signals `E1` to `m1()`, so the class handler `E1Handler()` of the `ExceptionalClient` is invoked. Case `m3()` signals `E4`, so the class handler `E4Handler()` is also invoked since `E4` is subtype of `E1` (Figure 4). If `m3()` signals `E5`, so the class handler `E5Handler()` inherited from `ExceptionalSupClient` is invoked. Case `m3()` signals `E6` to `m1()`, the object handler `E6Handler()` of `Exceptional_client1` (Figure 5) is invoked in spite of presence of the class handler. Supposing `m3()` is invoked by `m2()` of the object `client1`, if `m3()` signals `E6`, the method handler `m2E6Handler()` of `Exceptional_client1` is invoked.

5.3. Exception propagation

Our exception handling model defines explicit propagation of exceptions whose benefits are notably discussed in [3]. The handling of signaled exceptions is limited to the immediate caller. If a signaled exception is not handled in the caller, then the predefined exception failure is further propagated. However, the exception still may be ressignaled explicitly within a handler to a higher-level component. Despite gains in programming simplicity, the use of exceptions propagated automatically remains fault-prone because they are the least well documented and tested parts of an interface [4, 9]. The CLU designers [13] argue persuasively that this limitation supports the goals of good program structuring with only a minor loss in its writeability.

5.4. Continuation of control flow

We choose the termination model which consists of terminating the execution of the unit that raises the exception and then transferring control to the exception handler. The semantic of the termination model is more simple and suitable for construction of dependable systems [2]. Mechanisms that support resumption are very powerful and flexible, but it turns out to be difficult to use by application programmers. In fact, it can promote the unsafe programming practice of removing the symptom of an error without removing the cause.

5.5. Support to coordinated recovery

Since the cooperating activities are application-dependent, a support should be provided to application programmers in order to structure their cooperating tasks. In this work we apply a group framework as a means of allowing designers to improve the structuring of concurrent object-oriented systems, and supporting coordinated recovery. In this sense, coordinated recovery only needs be activated within the participant threads of a group. This obviously restricts system design but it makes possible to regard each group as a recovery region and attach fault tolerance activities to each group participant. We enable the subgroup's definition that contributes to control the system complexity and allows better organization of both normal and abnormal activities of the enclosing group.

Figure 6 shows threads that are represented as lines and activities of the groups are delimited by rectangles. The group `B`, composed by threads `T5`, `T7` e `T9`, is a subgroup of the group `A` which have the same composition of `B`, added of `T4`. After the occurrence of an exception in one of these threads (`T3`), other participants of the same group (`T4` e `T9`) should be notified in order to start forward error recovery. If any suitable handler has not been defined at least in one of the group participants, an abort exception is raised and then group activity must be undone (backward error recovery) and such an exception must be signaled for the enclosing group (group `A`). If the backward error recovery is not executed with success within the group then a failure exception is signaled to the enclosing group.

Each group has participants which are activated by some external activities, e.g. threads, and which cooperate within the group scope. Participants execute object methods that should have been designed to work cooperatively by means of shared objects. Participants may enter asynchronously in the group activity, but should exit in a synchronized way. Such participants may be
localized in different machines of a distributed environment. Each group participant has a set of exception handlers that are designed to recover the group cooperatively of eventual errors. An exception tree (Section 3.2) is associated to each group in order to do the resolution of the exceptions raised concurrently.

**Figure 6. Exception propagation**

**IMPLEMENATION OF COOPERATING THREAD GROUPS.** To implement cooperating thread groups, we provide two classes that can be used to define groups that need coordinated recovery. To implement a group, the first step is to define a class that extends the class `Group`. The class `Group` contains the methods to deal with the creation and the termination of each participant. Secondly, the programmer should define the participants that compose the group by extending the class `Participant`. Figure 7 shows the definition of a group (`Group1`) with two types of participants (`Participant1` and `Participant2`). Each class that derives from the class `Participant` should be instantiated (participant1 and participant2) before starting the group activity. To build an instance of this class, the object and the method that each participant executes should be passed as parameters. Such methods should have been designed for working cooperatively. A new class that extends the class `Group` also must be instantiated (`group1`) before starting the group activity. To build an instance of this class, should be passed the following parameters: (i) the set of group participants, (ii) the set of simple and structured exceptions which should be handled cooperatively by group participants, and (iii) the set of exceptions which should be signaled by the group for the external environment.

**Figure 7. The definition of a group**

The participants still may register themselves dynamically in a group through the method `RegisterParticipant` of the class `Group`. This class still provides the method `StartParticipant` (Participant) that allows a participant enters dynamically in a group activity. Participants finalize their participation in the group activity through the method `FinishParticipant` (Participant).

**IMPLEMENTATION OF SIMPLE AND STRUCTURED EXCEPTIONS.** The class `GroupException` should be used to define the exceptions that may be raised in cooperating thread groups and that need coordinated recovery. We adopt the Composite design pattern [6] (Figure 8) to define simple and structured exceptions. This pattern allows application designers treat simple exceptions and compositions of these (structured exceptions) uniformly. Simple exceptions are defined by extending the class `GroupException` (E1 and E2). Structured exceptions are instances of the class `StructuredException` (e12). The simple exceptions (E1 and E2) that compose a structured exception (e12) should be passed as parameters to create such a structured exception. Hence, each structured exception has a list of its constituent exceptions.

**Figure 8. The definition of the exceptions of a group**

6. Twin-engine aircraft control system

This Section highlights the benefits of the proposed exception mechanism for the design of reusable and dependable software. We present a twin-engine aircraft control system that is based on the example described in [1]. Consider a twin-engine aircraft control software that contains two components responsible for managing two engines: a left engine and a right engine. Such components can be defined as participant threads of a group; they cooperate for maintaining the aircraft stability.

**Figure 9. The activity of the group stability**

Figure 9 shows the participants left_engine and right_engine of the group stability. They cooperate through a shared object called state. Such an object is used by the participants to exchange information which is utilized, for instance, on the control adjustment. The exception tree for this group is shown in Figure 10. If the left (or right) engine fails, the left_engine (or right_engine) signals the
exception LeftException (or RightException) and handlers are activated in both participants. The handlers should adjust the controls appropriately to compensate for the loss of the left (right) engine in order to conduct the aircraft to the nearest airport. If both the right and left engine fail, the exceptions RightException and LeftException are raised concurrently by, respectively, left_engine and right_engine. The exception resolution procedure is accomplished by exception mechanism that searches the handlers for the structured exception emergency_exception within both participants. Immediately, the handlers are activated for this more serious exception. Such handlers should execute the emergency landing procedure. Besides, other exceptions could occur that would endanger the emergency landing procedure (for instance, fire). All such exceptions, if not listed individually within the exception tree, are categorized as the universal exception.

![Figure 10. The exception tree of the group stability](image)

Figure 10 shows a set of classes and their corresponding instances for the group Stability. The class Engine extends the class Participant and represents the group participants. The class Stability that derives from the class Group represents the group. To start group activity must be created two instances of the class Engine and one of the class Stability. Participants execute object methods for performing the cooperation group activity. In this example, the participants left_engine and right_engine execute the methods of the objects left_control and right_control when performing the cooperation group activity. The object called state, instance of the class State, is purposed for communication between the cooperating participants. The simple exceptions LeftException and RightException and the structured exception emergency_exception are also defined. The exceptional classes Exceptional_left_engine and Exceptional_right_engine contain the methods that are the handlers responsible for the coordinated recovery in participants left_engine and right_engine.

Object handlers should be defined for the group participants. Note that the classes Exceptional_left_engine and Exceptional_right_engine implement the handlers for all exceptions that can be raised by the participants. The structured exception can be defined by creating the following instance:

```
emergency_exception = new StructuredException (LeftException, RightException);
```

The set of initializations necessary for starting group activity can be as follows:

1. `Object[] Participants = {left_engine, right_engine};`
2. `Object[] InternalExceptions = {LeftException, RightException};`
3. `Object[] ExternalExceptions = {emergency_exception};`
4. `stability = new Stability (Participants, InternalExceptions, ExternalExceptions);`
5. `stability.StartAllParticipant();`

Line 1 creates the array with group participants. Line 2 creates the array with exceptions that may be raised and must be treated cooperatively by the group. Line 3 creates the array with exceptions that must be signaled by the group for external environment. Line 4 creates the object that represents the group Stability. Line 5 starts the group activity through the method stability.StartAllParticipant().

7. Implementation

In this Section, we present a meta-level software architecture that implements an exception mechanism. The architecture consists of a base level and a meta-level. The base-level objects are the objects of the application, while the meta-objects implement the specific responsibilities of the exception mechanism. When a base-level object signals an exception, it is intercepted by the MOP and the corresponding meta-object searches for an adequate handler in a transparent way to application (base level). Applications are composed of normal classes that implement the normal functionality and exceptional classes with handlers for the corresponding normal classes.

Figure 12 illustrates the meta-level architecture that implements the exception mechanism. The base level is composed of: (i) the exception class hierarchy (Figure 4); (ii) normal class hierarchies (Figure 3); (iii) exceptional...
classes with handlers that are associated to normal classes (Figure 3) and (iv) exceptional classes with handlers that are associated to objects (Figure 5).

The meta-level is composed of: (i) composers, and (ii) meta-searchers. The composers are special meta-objects associated to application’s objects or classes. They are important components because they allow delegate information and operations of these objects and classes to meta-objects responsible for several management actions, such as exception handling, persistency and atomic actions. The meta-searchers are meta-objects responsible for managing exception handling. Besides, they receive information and operations reified by the composers. Based on these operations and their results, the meta-searchers execute the following activities: (i) search for a suitable handler associated to the raised exception; (ii) invocation of the handler; (iii) return to the normal operation of the application.

Figure 12. The proposed meta-level architecture

7.1. Concurrency

Figure 13 shows the components of the meta-level architecture that implements cooperating thread groups. The meta-level is composed of the following components: (i) composers, (ii) meta-searchers, (iii) meta-groups, and (iv) EPS (Event Processor Service) [17]. Each instance of Participant (Section 5.5) is associated to a composer and a meta-searcher; each instance of Group (Section 5.5) is associated to a composer and a meta-group. The composers and meta-searchers were previously described. Meta-groups are meta-objects responsible for managing the coordinated recovery for exceptions raised by cooperating thread groups. Meta-groups hold the following meta-information: (i) the set of group participants, (ii) the set of simple and structured exceptions which should be handled cooperatively by group participants, and (iii) the set of exceptions which should be signaled by the group for the external environment. The meta-group sends the simple and structured exceptions to EPS that must compose the group exception tree. EPS is a monitor for distributed and composite events which is able to process generic events. In this work, EPS is an application utilized for monitoring exceptions that may be raised concurrently in cooperating thread groups. EPS and meta-group accomplish the exception resolution procedure. When an exception occurs, EPS informs the meta-group, which in turn informs the participants and coordinates the invocations of the handlers in order to start the coordinated recovery. Therefore handlers are activated in a transparent way to application.

Figure 13. The meta-level architecture for concurrency

7.2. Implementation issues

Our mechanism does not require any special language support, and it was implemented within the Java programming language. We utilized Java Remote Method Invocation (RMI) API in order to distribute the software components through a network. Moreover, the EPS has allowed the construction of the exception tree composition scheme based on the aggregated tree concept [17], which has ensured gains in performance.

Our mechanism was implemented without any changes to the language itself by means of a meta-object protocol called Guaraná [18]. Guaraná is a flexible meta-object protocol for Java language that allows creating meta-level objects. Guaraná provides an efficient broadcast service for communication between meta-objects. Moreover, it provides support to compose meta-objects responsible for different management functions through the composers. These Guaraná capacities and how our exception mechanism was designed allow that the meta-objects of our exception handling system be easily integrated with meta-objects responsible for other administrative (non-functional) services, such as persistency and atomic actions.

8. Related Work

The work of Hof et al. [8] describes an exception mechanism based on meta-programming and computational reflection. Their implementation was carried out in a specific system but it could be ported to most other systems that support meta-programming. However, such a mechanism does not support coordinated recovery in concurrent threads and is not integrated with
the object paradigm. Arche language [10] allows user-defined resolution of multiple exception amongst a group of objects that belong to different implementations of a given type; however, this approach is not generally applicable to the coordinated recovery of multiple interacting objects of different types. In our exception handling model, coordinated recovery can be applied to a group of interacting objects of different types.

9. Concluding remarks and future work

The current lack of effective error-handling techniques for constructing dependable object-oriented software motivated us to develop the design and implementation of an object-oriented exception mechanism. Our exception handling model supports a clear and transparent separation between the normal and exceptional activities of a software component. This separation has allowed to produce software components which are easy to understand, to change and to maintain in the presence of faults. Exceptional classes allow the uniform and non-intrusive implementation of error-handling code for every kind of component (concurrent or not). The exceptional class hierarchy allows exceptional code reuse. Moreover, our mechanism is integrated with object paradigm and provides support to coordinated recovery.

Our mechanism does not require any special language support, and it was implemented within the Java programming language without any changes to the language itself. The implementation of a meta-level architecture allowed the separation between activities related to management of the exception handling from the exceptional and normal activities of the application.

The coordinated atomic action concept (CAAction) [19] was introduced as a unified approach for structuring complex concurrent activities and supporting error recovery between multiple interacting objects in a distributed object-oriented system. We plan to integrate the proposed exception mechanism within a CAAction framework.

Nowadays off-the-shelf approach to object-oriented software development, achieved by selecting and configuring reusable components, has resulted in significant decrease of development cost. In this work, we have designed a mechanism that supports the construction of reusable and dependable software components. However, an open issue is how to allow that exception-handling code be added on reusable components (for instance, COTS) without any interference on the original code of these components. This additional exception-handling code should handle the new exceptions that can arise when these components are reused on different applications.

Acknowledgements

This work has been supported by CNPq/Brazil under grant n° 131945/98-0 for Alessandro F. Garcia, grant n° 141425/97-0 for Delano M. Beder and grant n° 351592/97-0 for Cecilia M. F. Rubira.

References