Tratamento de Exceções em Sistemas Concorrentes Orientados a Objetos

Alessandro Fabricio Garcia

Dissertação de Mestrado

Tratamento de Exceções em Sistemas Concorrentes Orientados a Objetos

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Alessandro Fabricio Garcia e aprovada pela Banca Examinadora.

Campinas, 14 de abril de 2000.

Cecília Mary Fischer Rubira

Cecília Mary Fischer Rubira (Orientadora)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Instituto de Computação
Universidade Estadual de Campinas

Tratamento de Exceções em Sistemas Concorrentes Orientados a Objetos

Alessandro Fabricio Garcia[1]

Março de 2000

Banca Examinadora:

- Cecília Mary Fischer Rubira (Orientadora)
- Prof. Paulo Henrique Monteiro Borba
  Centro de Informática (CIn-UFPE)
- Profa. Eliane Martins
  Instituto de Computação (IC-UNICAMP)
- Prof. Luiz Eduardo Buzato (Suplente)
  Instituto de Computação (IC-UNICAMP)

[1]**Supported by CNPq, grant 131945/98-0**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Garcia, Alessandro Fabricio

G165t Tratamento de exceções em sistemas concorrentes orientados a objetos / Alessandro Fabricio Garcia-- Campinas, [S.P. :s.n.], 2000.

Orientador : Cecília Mary Fischer Rubira

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Tolerância a falhas. 2. Linguagem de programação (Computadores). 3. Engenharia de software. I. Rubira, Cecília Mary Fischer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 14 de abril de 2000, pela Banca Examinadora composta pelos Professores Doutores:

Prof. Dr. Paulo Henrique Monteiro Borba
UFPE

Profa. Dra. Eliane Martins
IC-UNICAMP

Profa. Dra. Cecilia Mary Fischer Rubira
IC-UNICAMP

"Aos meus pais Antonio e Cida
e a minha irmã Andresa pelo amor e incentivo prestados incondicionalmente ao longo de toda minha vida."

Agradecimentos

Este trabalho é fruto de muito esforço e dedicação dispensados nesses dois anos de mestrado. A felicidade é imensurável quando percebo quantas alegrias foram vivenciadas e quantos desafios foram superados nesses anos. Certamente, é a Deus que manifesto minha eterna gratidão e dedico todas essas alegrias e vitórias.

Ilimitados agradecimentos vão para as três pessoas que julgo serem as responsáveis primeiras por tudo que alcancei até aqui: meus preciosos pais Antonio e Cida e minha irmã Andresa. O apoio constante da minha família foi essencial para encontrar forças e continuar a lutar pelos meus objetivos.

Repleto de alegria pela fidelidade prestada, venho agradecer com toda sinceridade, aos meus amigos, aqueles verdadeiros, que me apoiaram incansavelmente nos momentos de alegria e de dificuldade. Não referencio todos os nomes aqui, pois a lista seria enorme e também para não pecar pelo esquecimento. Caros amigos, por tudo que realizamos e por tudo que ainda realizaremos juntos: muito obrigado!

Agradeço a minha orientadora Cecília Rubira pelos inúmeros ensinamentos que me transmitiu durante o nosso convívio, os quais foram fundamentais para a realização deste trabalho. Agradeço também aos demais professores e funcionários do Instituto de Computação.

Finalmente, agradeço ao CNPq pelo apoio financeiro.

Resumo

Sistemas orientados a objetos confiáveis devem incorporar atividades de tratamento de exceções de forma a comportarem-se adequadamente sob uma grande variedade de situações, inclusive na presença de erros. Nesse contexto, um mecanismo de tratamento de exceções é fundamental para detecção e recuperação de erros bem como para ativação das medidas necessárias para restaurar a atividade normal do sistema. O desenvolvimento de um mecanismo de tratamento de exceções é uma tarefa difícil especialmente quando a concorrência é uma das características dos sistemas de software. O principal objetivo desta dissertação é o projeto e implementação de um mecanismo de tratamento de exceções para a construção de sistemas orientados a objetos confiáveis. Na construção do mecanismo proposto, nós utilizamos técnicas de estruturação de software, tais como reflexão computacional e padrões de projeto. Duas contribuições são consideradas principais. A primeira delas, caracterizada por aspectos técnicos e usos práticos, é o projeto e implementação de um mecanismo de exceções utilizando a linguagem de programação Java e uma arquitetura de software reflexiva chamada Guaraná. O mecanismo proposto especialmente oferece suporte a tratamento de exceções concorrentes. A outra contribuição, caracterizada por aspectos abstratos e abordagem inovadora, é a definição de uma arquitetura de software reflexiva e um conjunto de padrões de projeto relacionados para a implementação de mecanismos de tratamento de exceções.

Abstract

Dependanble object-oriented software should incorporate exception handling activities in order to behave suitably in a great number of situations in spite of errors. In this context, an exception handling mechanism is fundamental to detect errors, and to activate the suitable measures to restore the normal activity of the system. The development of an exception handling mechanism is not a trivial task. This task is specially difficult when the software using the exception mechanism is concurrent. The main aim of this work is to propose the design and implementation of an exception handling mechanism for developing dependable object-oriented software. In order to build the proposed mechanism we apply techniques of software structuring, such as computational reflection and design patterns. The main contribution of this work is the design and implementation of an exception handling mechanism using the Java language and a reflective software architecture called Guaraná. The proposed mechanism specially supports concurrent exception handling. In addition, we define a reflective software architecture and a set of design patterns for implementing exception handling mechanisms.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução Geral

O uso crescente de sistemas computacionais em quase todos os ramos da sociedade tem levado a necessidade de desenvolvimento de sistemas de software confiáveis. O paradigma de objetos é uma das formas promissoras para construção de software de qualidade e pode contribuir decisivamente para a prevenção e remoção de falhas durante as fases do ciclo de desenvolvimento de software. Entretanto, a presença de falhas residuais é inevitável mesmo em sistemas orientados a objetos devido a complexidade inerente aos sistemas de software atuais. Essas falhas podem ocasionar efeitos indesejáveis no sistema durante sua vida operacional. O desenvolvimento de sistemas orientados a objetos confiáveis não é uma tarefa trivial. Projetistas de sistemas confiáveis devem lidar com as situações excepcionais possíveis e incorporar ao sistema atividades de tolerância a falhas capazes de evitar um defeito catastrófico. As atividades de tolerância a falhas introduzidas usualmente aumentam a complexidade do sistema de software. Nesse contexto, um mecanismo de tratamento de exceções é fundamental para detecção e recuperação de erros, causados por falhas residuais, e para a estruturação e ativação das medidas apropriadas de tolerância a falhas de forma a restaurar a atividade normal de um sistema de software confiável.

Programadores utilizam mecanismos de tratamento de exceções (mecanismo de exceções) para a implementação das atividades de tratamento de exceções (erros) de um sistema de software. As atividades de tratamento de exceções implementam as medidas para tolerar as falhas que podem se manifestar durante a execução da atividade normal. O mecanismo de exceções é responsável pela interrupção do fluxo normal do sistema quando a ocorrência de uma exceção é detectada durante a sua execução, e a ativação das medidas de tolerância a falhas adequadas.

1.1 O Problema

O desenvolvimento de mecanismos de exceções adequados para a construção de software orientado a objetos confiável implica em um grande desafio. Estes mecanismos devem prover suporte para uma separação explícita entre as atividades normais e as atividades de tratamento de exceções de tal forma a manter sob controle a complexidade geral de sistemas confiáveis. Esses mecanismos também devem ser simples, restritivos e integrados com o paradigma de objetos. Mecanismos baseados em soluções complexas e com flexibilidade desnecessária proporcionam a introdução de erros adicionais ocasionados pelo seu uso. Além disso, o desenvolvimento de um mecanismo de exceções sofre impactos adicionais quando a concorrência é uma das características dos sistemas de software.

Tratamento de exceções é muito mais difícil em sistemas concorrentes devido a concorrência cooperativa [38]. Em particular, para sistemas orientados a objetos, processos concorrentes podem cooperar através de comunicação inter-processos para a realização de alguma atividade do sistema. Eventualmente, um dos processos pode levantar uma exceção. Essa exceção não pode ser tratada isoladamente no processo que a levantou, uma vez que informação errônea pode ter sido espalhada através de comunicação inter-processos. Além disso, devido a própria natureza de software concorrente, mais de uma exceção pode ser levantada mais ou menos ao mesmo tempo nos diferentes processos. O levantamento de múltiplas exceções concorrentemente pode ser o sintoma de uma falha mais séria [9]. Nesses casos, o mecanismo de exceções deve dar suporte à recuperação de erros de forma coordenada entre os processos envolvidos na cooperação. Todos processos participantes da atividade cooperativa devem ser informados da ocorrência de exceções e devem ser envolvidos no processo de recuperação de erros.

1.2 Limitações das Soluções Existentes

Mecanismos de exceções são usualmente considerados como uma parte essencial de qualquer linguagem de programação moderna e, tipicamente, modelos distintos de tratamento de exceções são adotados para o projeto desses mecanismos nas diferentes linguagens. Entretanto, um estudo comparativo de mecanismos existentes em linguagens orientadas a objetos mostrou que eles não satisfazem algumas características desejáveis [25] (Capítulo 2). Desenvolvedores desses mecanismos se preocupam geralmente em prover soluções amplamente flexíveis sem a atenção devida para simplicidade e outros requisitos importantes. Os mecanismos existentes muitas vezes não possuem um projeto orientado a objetos e provêem suporte limitado para uma separação explícita entre as atividades normais e excepcionais de uma aplicação.

Uma das principais deficiências dos mecanismos de exceções disponíveis é a inexistência

de suporte apropriado para tratamento de exceções concorrentes. Os mecanismos existentes geralmente são dedicados para programas sequenciais. Somente a linguagem Arche [33] provê um esquema para tratamento de condições excepcionais em sistemas concorrentes. Entretanto, o modelo de concorrência implementado em Arche limita-se a ativar recuperação de erros entre objetos do mesmo tipo. Assim, as linguagens de programação orientadas a objetos atuais não provêem, de forma satisfatória, mecanismos de tratamento de exceções adequados para o desenvolvimento de sistemas orientados a objetos confiáveis.

Recentemente, alguns trabalhos [58, 60, 74] têm proposto mecanismos dedicados especialmente para tratamento de exceções concorrentes como extensões para determinadas linguagens de programação específicas. As abordagens propostas exigem alguma modificação da linguagem de programação e/ou de seu compilador ou interpretador, ou a definição de uma interface de programação para tratamento de exceções concorrentes. Entretanto, estas abordagens usualmente apresentam soluções complicadas e implementam um modelo de tratamento de exceções não integrado com o modelo de objetos. Ademais, tais propostas são intrusivas do ponto de vista da aplicação uma vez que o código da aplicação é embutido com uma série de chamadas a serviços específicos do mecanismo de tratamento de exceções concorrentes. O código extra inserido dificulta a legibilidade, a reutilização e a manutenção dos componentes da aplicação. Consequentemente, as abordagens existentes propõem mecanismos para tratamento de exceções concorrentes que são difíceis de usar, e usualmente conduzem ao desenvolvimento de sistemas orientados a objetos não confiáveis e que são difíceis de entender, manter e reutilizar.

1.3 Objetivos

Em resumo, os principais objetivos desta dissertação são:

1. Proposta de projeto e implementação de um mecanismo orientado a objetos de tratamento de exceções para o domínio de sistemas orientados a objetos confiáveis.

2. Utilização prática de técnicas avançadas de estruturação de software para a construção do mecanismo proposto, tais como reflexão computacional e padrões de projeto, e a análise das vantagens e limitações destas técnicas no desenvolvimento do mecanismo de exceções.

1.4 A Solução Proposta

Este trabalho apresenta o projeto e implementação de um mecanismo de exceções apropriado para a construção de software orientado a objetos confiável. O mecanismo incor-

pora um modelo orientado a objetos de tratamento de exceções e permite uma separação explícita entre as atividades normais e excepcionais de um sistema. O modelo proposto especialmente proporciona suporte para tratamento de exceções concorrentes. Técnicas de estruturação de software são utilizadas para a construção de um mecanismo de exceções que seja de fácil uso e reutilização.

A técnica de reflexão computacional é utilizada para implementação do modelo proposto de tratamento de exceções. O uso dessa técnica permite a introdução do mecanismo de exceções para a linguagem de programação sem criar modificações para a própria linguagem. Além disso, a utilização de reflexão computacional permite uma melhor divisão entre a funcionalidade da aplicação e os serviços específicos do mecanismo de exceções. Essa divisão alcançada permite a obtenção de um mecanismo de exceções não intrusivo e fácil de usar. O mecanismo proposto está implementado na linguagem Java e usa uma arquitetura de software reflexiva para esta linguagem chamada Guaraná [51].

Nós também definimos o projeto de uma arquitetura de software reflexiva para mecanismos de tratamento de exceções. A arquitetura é descrita por um conjunto de componentes com responsabilidades bem definidas e a interação entre esses componentes. Esta arquitetura oferece uma solução de projeto genérica que integra uniformemente tratamento de exceções para programas sequenciais e concorrentes. A arquitetura proposta é descrita de forma independente de linguagem de programação e pode ser reutilizada em diferentes aplicações. Um conjunto de quatro padrões de projeto documenta os aspectos estruturais e comportamentais dos componentes da arquitetura de software proposta. Padrões de projeto constituem boas soluções de projeto para problemas recorrentes dentro de um contexto particular [7, 20]. No contexto deste trabalho, os padrões propostos incorporam boas soluções conhecidas para os problemas comuns no domínio de mecanismos de exceções.

1.5 Contribuições

Este trabalho apresenta as seguintes contribuições:

1. Um estudo comparativo dos diferentes modelos de tratamento de exceções implementados em diversas linguagens orientadas a objetos e proposta de uma taxonomia que permite avaliá-los.

2. Proposta de um critério de projeto com os requisitos desejáveis para mecanismos de tratamento de exceções que serão utilizados na construção de sistemas orientados a objetos confiáveis. Um modelo ideal de tratamento de exceções é proposto tendo como base o critério de projeto definido. O modelo proposto especialmente dá suporte a tratamento de exceções concorrentes.

3. Projeto e implementação de um mecanismo de exceções para a linguagem Java utilizando a arquitetura de software reflexiva do Guaraná. O mecanismo implementa o modelo proposto de tratamento de exceções que contempla o critério de projeto definido.

4. Definição de uma arquitetura de software reflexiva para o projeto de mecanismos de exceções que serão utilizados na construção de sistemas orientados a objetos confiáveis.

5. Proposta de um conjunto coeso de quatro padrões de projeto que documentam os aspectos de estrutura e comportamento dos componentes arquiteturais de um mecanismo de exceções, e incorporam boas soluções conhecidas para os problemas comuns no domínio desses mecanismos.

1.6 Organização da Dissertação

Esta dissertação é uma coleção de artigos científicos escritos em inglês que foram publicados ou submetidos para publicação em simpósios e revistas internacionais. O restante desta dissertação está organizada da seguinte forma:

Capítulo 2 contém o artigo *"A Comparative Study of Exception Handling Proposals for Dependable Object-Oriented Software"* [25]. Este artigo apresenta a terminologia relacionada a tratamento de exceções e tolerância a falhas utilizada neste trabalho, bem como discute as dificuldades relacionadas a tratamento de exceções concorrentes. Este artigo também revisa diferentes modelos de tratamento de exceções implementados em diversas linguagens orientadas a objetos e propõe uma taxonomia. A taxonomia desenvolvida permite classificar e comparar os modelos de tratamento de exceções estudados. Finalmente, este artigo apresenta um critério de projeto adequado para mecanismos de exceções bem como um modelo ideal de tratamento de exceções.

Capítulo 3 contém o artigo *"An Exception Handling Mechanism for Developing Dependable Object-Oriented Software Based on a Meta-Level Approach"* [21]. Este artigo apresenta o projeto e implementação de um mecanismo de tratamento de exceções utilizando a arquitetura de software reflexiva do Guaraná [51]. O mecanismo implementa o modelo proposto de tratamento de exceções que contempla o critério de projeto definido no Capítulo 2.

Capítulo 4 contém o artigo *"An Exception Handling Software Architecture for Developing Robust Software"* [22]. Este artigo define uma arquitetura de software reflexiva para mecanismos de tratamento de exceções que serão utilizados na construção de siste-

mas orientados a objetos confiáveis. Além disso, este artigo propõe os padrões de projeto que documentam a estrutura e o comportamento dos componentes arquiteturais de um mecanismo de tratamento de exceções.

Capítulo 5 resume as conclusões do nosso trabalho, apresentando as principais contribuições e os possíveis trabalhos futuros.

Capítulo 2

Um Estudo Comparativo de Mecanismos de Exceções

Mecanismos de tratamento de exceções são usualmente considerados como uma parte essencial de qualquer linguagem de programação orientada a objetos. No contexto de sistemas orientados a objetos confiáveis, mecanismos de exceções são usados para detecção e recuperação de erros, e para estruturar as atividades de tolerância a falhas incorporadas em um sistema. Modelos distintos de tratamento de exceções são adotados para o projeto desses mecanismos nas diferentes linguagens. Um mecanismo de exceções para software orientado a objetos confiável deve incorporar um modelo adequado de tratamento de exceções.

Este capítulo contém o artigo *"A Comparative Study of Exception Handling Proposals for Dependable Object-Oriented Software"* [25], que foi submetido para a revista *"Journal of Systems and Software"*. Este artigo apresenta os conceitos relacionados a tratamento de exceções e tolerância a falhas utilizados neste trabalho, bem como discute as dificuldades relacionadas a tratamento de exceções concorrentes. Este artigo também revisa diferentes modelos de tratamento de exceções implementados em diversas linguagens orientadas a objetos e propõe uma taxonomia. A taxonomia desenvolvida é utilizada para classificação e comparação dos modelos de tratamento de exceções estudados. Finalmente, este artigo apresenta um critério de projeto adequado para mecanismos de tratamento de exceções e um modelo ideal de tratamento de exceções.

A Comparative Study of Exception Handling Proposals for Dependable Object-Oriented Software

Alessandro F. Garcia **Cecília M. F. Rubira**
Institute of Computing
University of Campinas (UNICAMP)
Campinas – Brazil
{afgarcia, cmrubira}@dcc.unicamp.br

Alexander Romanovsky
Department of Computing Science
University of Newcastle upon Tyne
Newcastle upon Tyne – United Kingdom
{alexander.romanovsky}@newcastle.ac.uk

Jie Xu
Department of Computer Science
University of Durham
Durham – United Kingdom
{Jie.Xu}@durham.ac.uk

2.1 Introduction

Dependable object-oriented systems have to cope with a number of exceptional situations and incorporate fault tolerance activities in order to meet the system's robustness and reliability requirements. With such systems growing in size and complexity, employing error-handling techniques and satisfying the requirements of software qualities such as maintainability and reusability are still deep concerns to engineers of dependable object-oriented systems. Exception handling mechanisms are the most important schemes for detecting and recovering errors, and structuring the fault tolerance activities incorporated in a system. However, the current lack of suitable exception handling mechanisms can make an application non-reliable and difficult to understand, maintain and reuse in the presence of faults.

Engineers of dependable object-oriented systems utilize exception mechanisms to de-

velop exception handling activities for dealing with such erroneous situations. In this systems, the code devoted to error detection and handling is usually both numerous and complex. As a consequence, up to two-thirds of a program can be for error handling [11, 26]. In this context, the design of an exception mechanism should be simple and easy to use, and provide explicit separation between the normal and exceptional code. Ideally, dependable object-oriented systems using the exception mechanism should be easy to understand, maintain and reuse. A number of exception mechanisms have been developed to object-oriented programming languages. Realistic examples of object-oriented languages include Java [30], Modula-3 [50] and Eiffel [44].

The purpose of our paper is to investigate the applicability of the existing exception mechanisms of object-oriented programming languages for developing dependable object-oriented software with effective quality attributes. The major contributions of this article are: (i) the definition of a set of adequate design solutions while developing an exception mechanism suitable for dependable object-oriented software, (ii) the presentation of a comprehensive survey of existing exception mechanisms implemented in object-oriented languages, (iii) comparison and evaluation of the investigated mechanisms as well as the identification of the primary limitations in applying them in practice to develop dependable object-oriented systems, and (iv) the identification of current trends related to exception handling and dependable object-oriented software. A taxonomy is used to discuss nine functional aspects of an exception mechanism and to distinguish one mechanism from another - especially support for concurrent exception handling is examined in detail.

The remainder of this article is organized as follows. Section 2 gives a brief description of exception handling within a framework for facilitating software fault tolerance. This section also introduces exception mechanisms as well as difficulties related to concurrent exception handling. Section 3 describes our proposed taxonomy for classifying different design approaches to exception mechanisms. Section 4 presents a general criteria to design an effective exception mechanism for developing dependable object-oriented systems. Section 5 discusses in more detail exception models implemented in various object-oriented languages. Section 6 assesses the relative advantages, disadvantages and general limitations of these models based on our established design criteria. Section 7 discusses difficulties and directions for future work. Finally, Section 8 presents some concluding remarks.

2.2 Exception Handling and Fault Tolerance

2.2.1 Terminology

Following the terminology adopted by Lee and Anderson [38], 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. Dependable software systems require supplementary techniques in order to tolerate the manifestations of faults in its components and, consequently, to avoid failures of the system. In general, these techniques are based on the provision of redundancy which increases the size of the systems and introduces additional complexity to them. Dependable software and its components should therefore be well-structured in order to master such an additional complexity. Each system component should be able to return well-defined responses, and incorporate a clear separation between normal and fault tolerance activities. In this sense, exceptions and exception handling provide a suitable framework for structuring the fault tolerance activities incorporated in a system.

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*. 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 cannot provide its specified service; (iii) *internal exceptions* which are exceptions raised by the component in order to invoke its own internal fault tolerance activity. The activity of a component can be divided in two parts: *normal activity* and *abnormal* (or *exceptional*) *activity* (Figure 2.1). The normal activity implements the component's normal services while the exceptional activity provides measures for tolerating the faults that cause exceptions. Thus, the normal activity of the system is clearly distinguished from its exceptional activity. At each level of the system, an *idealized fault-tolerant component* handles the exceptions raised during its normal activity and exceptions signaled by lower-level components. Whenever an exception is raised in a (server/callee) component that cannot handle it, the exception is signaled to the (client/caller) higher-level component that dynamically invoked the server component. After the exception is handled, the system may return to its normal activity.

Developers of dependable systems usually refer to faults as exceptions because they are expected to manifest rarely during the component's normal activity. Exception handling mechanisms (or merely exception mechanisms) have been developed for programming languages and allow software developers to define exceptions and to structure the exceptional activity of software components by means of handlers. The handlers of a program consti-

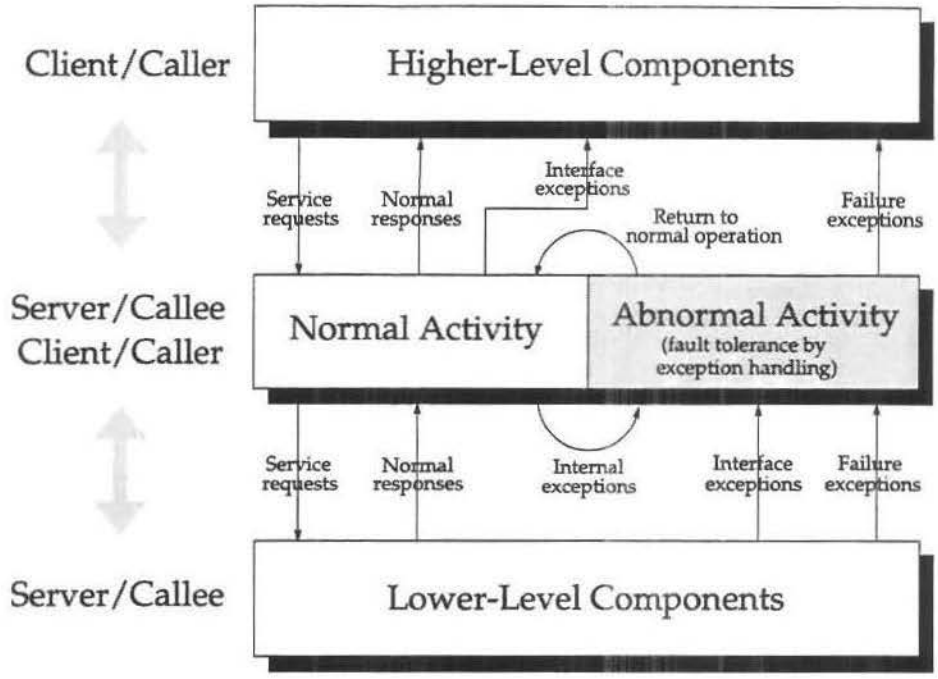


Figura 2.1: Idealized Fault-Tolerant Component

tutes its exceptional activity part. The exception mechanism is responsible for changing the *normal control flow* of a program to the *exceptional control flow* when an exception is raised during its execution. Exception mechanisms are either built as an inherent part of the language with its own syntax, or as a feature added on the language through library calls [17].

In the context of programming languages, exceptions are usually classified into two types [29, 37]: (i) *user-defined*, and (ii) *predefined*. User-defined exceptions are defined and detected at the application level. Predefined exceptions are declared implicitly and are associated with conditions that are detected by the language's run-time support, the underlying hardware or operating system. The kinds of exceptional events supported by a particular exception mechanism differ from one language to another and depend on general decisions taken by the language designers.

An exception can be raised at run-time (an *exception occurrence*) during the normal execution of an operation (method). A *signaling statement* is the statement being executed when an exception occurrence is detected. The code block containing the signaling statement is referred to as the exception *signaler* (Figure 2.2). When an exception occurrence is detected, the exception mechanism is responsible for searching and invoking an *exception handler* (or simply *handler*) to deal with the raised exception. The handler is the part of application's code that provides the measures for handling the raised exception. Some extra-information about an exception occurrence, such as its name, description, lo-

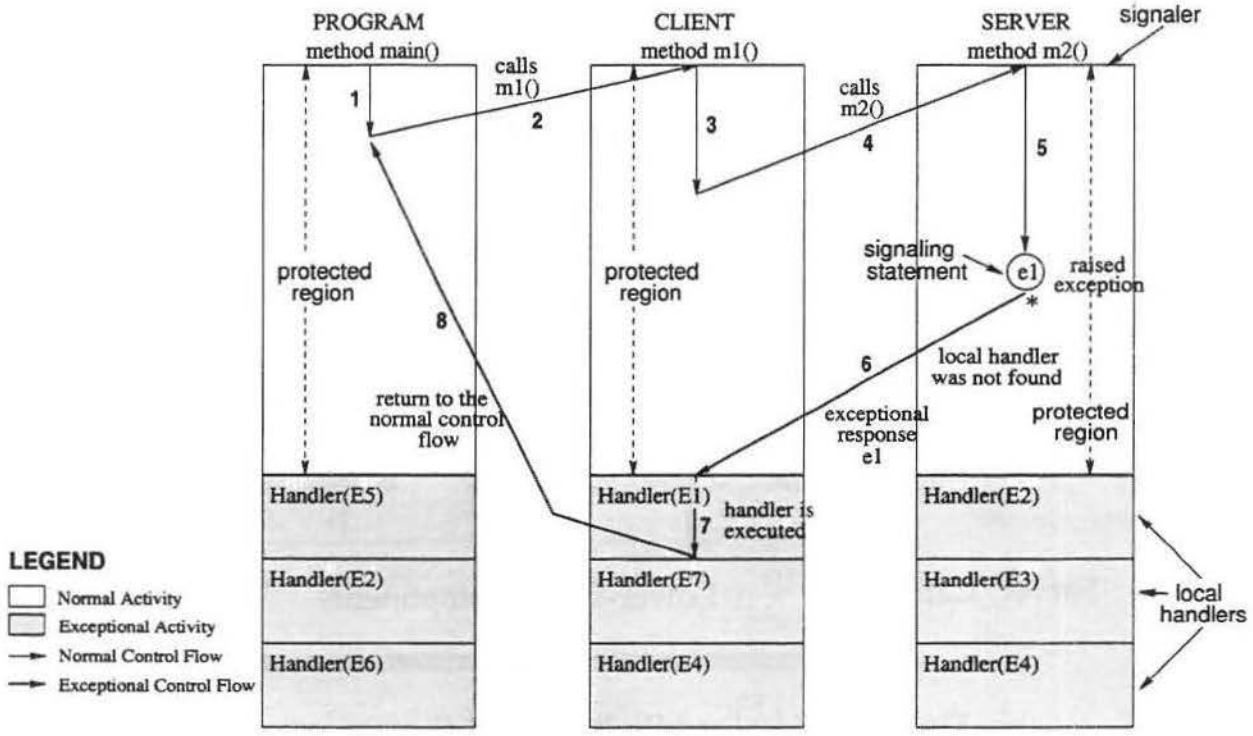


Figura 2.2: The Operation of an Exception Mechanism

cation, and severity [37], is usually required by the corresponding handler, and it is useful for handling an exception occurrence. Extra-information is passed either explicitly by the signaler, or implicitly by the exception handling mechanism.

Handlers are attached to a particular region of the normal code which is termed *protected region* or *handling context*. Figure 2.2 illustrates three protected regions. Each protected region can have a set of attached handlers. If an exception is raised in an protected region, the normal control flow is deviated to the exceptional control flow. Then the exception mechanism first tries to find a *local handler* attached to the protected region (the signaler). If it does not find a local handler for that exception, it searches the handlers provided by the enclosing protected region. If it again does not find an appropriate handler, the exception is signaled to the operation caller and this sequence of steps is again repeated. After a suitable handler is found, invoked and executed by the mechanism, it returns the computation to the normal control flow. In Figure 2.2, an exception e1 is raised during the execution of m2. The exception mechanism signals e1 to the caller, the method m1, since a local handler was not defined at the context of the signaler (arrow 6). The exception mechanism then finds and invokes the appropriate handler at the context of the caller (arrow 7), and returns the system to the normal control flow (arrow 8).

2.2.2 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 methods on objects (which are the subjects of computation). Exception handling is an important mechanism for achieving fault tolerance in sequential object-oriented software. Exception handling, and consequently the provision of fault tolerance, are much more difficult in concurrent object-oriented systems than in sequential ones. Exception mechanisms used in sequential programs cannot be applied to concurrent software without appropriate adjustments due to new difficulties introduced by concurrent exception handling.

From the standpoint of fault tolerance, the implementation of an exception mechanism for concurrent object-oriented systems is an interesting challenge due to cooperative concurrency [9, 38]. Different threads of a system can be cooperating for executing some system's task. 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 [38]. Erroneous information may spread directly or indirectly through inter-thread communication. As a consequence, the handling of an exception should involve all concurrent threads participating in a cooperation. Sometimes it may involve the entire system due to complex interactions between its cooperating threads. The cooperating threads of a concurrent system must be controlled very carefully in order to avoid that erroneous information spreads unexpectedly through the whole system [9].

The approach to using exception handling in such systems extends the well-known atomic action paradigm [9]. Atomic actions are the most comprehensive way of structuring the behavior of concurrent systems. These actions are units of inter-thread cooperation and their execution is indivisible and invisible from the outside. The activity of a group of threads participating in a cooperation constitutes an atomic action if there are no interactions between that group and the rest of the system for the duration of the activity [38]. Complex interactions between the participating threads of an atomic action can be coordinated within that action, including necessary activities for concurrent exception handling [9]. When one of the cooperating threads has raised an exception, error recovery should proceed in a coordinated way by triggering different handlers for the same exception within all the threads [9, 69].

An atomic action is formed by a group of cooperating threads, the *action participants*. The participants cooperate within the scope of the action. A set of exceptions is associated with each action. Each participant in the action has a set of handlers for (all or part of) these exceptions. The entries of participants in the action may be asynchronous but they have to leave the action synchronously to guarantee that no information is smuggled to or

from the action. When an exception has been raised in any of the participants inside an action, all action participants have to participate in the error recovery. Different handlers for the same exception have to be called in all of the participants [9]. These handlers cooperate to recover the action. The participants can leave the action on three occasions. First of all, this happens if there have been no exceptions raised. Secondly, if an exception had been raised, and the called handlers have recovered the action. Thirdly, they can signal a *failure* exception to the containing action if an exception has been raised and it has been found that there are no appropriate handlers or that recovery is not possible.

Furthermore, due to nature of concurrent systems, it is possible that various exceptions may be raised concurrently by cooperating threads. In this way, a mechanism for *exception resolution* is an essential part of concurrent exception handling. The paper [9] describes a model for exception resolution called *exception tree* which includes an exception hierarchy. This model allows to find the exception that represents all exceptions raised concurrently. This tree includes all exceptions associated with the action and imposes a partial order on them in such a way that a higher-level exception has a handler capable of handling any lower-level exception. If several exceptions are raised concurrently, the resolved exception is the root of the smallest subtree containing all of the exceptions.

A demand for concurrent exception handling in object-oriented systems is recognized by many researchers because it would make the error handling simpler, uniform and less error prone. Different works have identified the need for concurrent exception handling in a number of practical examples and systems in different application areas such as banking [22, 70], office automation [70], sales control systems [69], software development environments [70], and production cell control systems [62, 74, 55].

2.3 A Taxonomy for OO Exception Mechanisms

There is a number of design issues for building exception mechanisms that will be used for constructing dependable object-oriented software. However, the chosen solution for designing each of them varies from mechanism to mechanism. This section presents a taxonomy which identifies the several common design issues of exception mechanisms, and classifies the different design solutions. The taxonomy was developed based on the set of analyzed exception mechanisms (Section 2.4), and on some reviewed previous studies [37, 17].

We classify the design issues of an exception handling scheme into nine aspects of interest: (i) exception representation, (ii) exception interface, (iii) handler attachment, (iv) handler binding, (v) exception propagation, (vi) continuation of the control flow, (vii) cleanup actions, (viii) reliability checks, and (ix) concurrent exception handling. In the following we discuss each aspect in turn.

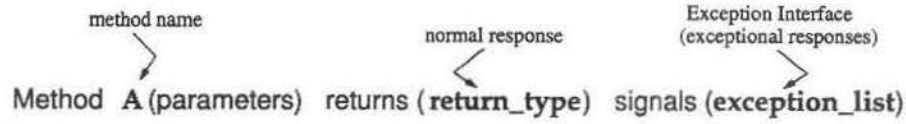


Figura 2.3: A Method's Signature with Exception Interface

Exception Representation. Exceptions that can be raised during a system's execution must be represented internally within this system. Exceptions can be represented as: (i) *symbols*, (ii) *data objects*, or (iii) *full objects*. The representation of exceptions as symbols is a classical approach in which exceptions are strings or numbers. Raising an exception sets the corresponding string variable (or integer variable) and returns the control to the caller that is in charge of testing the variable.

In the second and third solutions, exceptions are organized hierarchically as classes; when an exception is raised, an instance of an *exception class* (an *exception object*) is created and passed as a parameter to the corresponding handler. Therefore, exceptions are first-class objects. However, such solutions differ in how exceptions are raised. In the second solution, exceptions are raised by calling a keyword of the language. In the third solution, exceptions are raised by invoking a method raise on the exception object. In this last case, the exception is a standard object that receives messages since the behavior specific to exception raising is defined as a method on an exception class [36]. In addition, specific behaviors to continuation of the control flow (see below) can also be defined as methods on exception classes. Note that in the second solution the aim of exception objects is just to hold data, despite the possible definition of methods on them.

Exception Interface. A method may either terminate normally or exceptionally by signaling an exception. Exception interface is the part in a method's signature that explicitly declares the list of exceptions that might be signaled by the method [37] (Figure 2.3).

There are different design solutions for exception interface in different exception mechanism proposals. In some exception mechanisms, exception interfaces are *obliged* – an attempt to propagate to the invoker an exception that is not in the exception list causes either a compiling error or the raising of a predefined exception at run-time. In other mechanisms, exception interfaces either are *optional* or *unsupported*. There is also a *hybrid* approach – some exceptions must be listed in exception interfaces while others may not be listed.

Handler Attachment. Protected region is a domain that specifies the region of computation during which, if an exception occurrence is detected, a handler attached to this region will be activated. Handlers can be attached to different protected regions, such as:

(i) a *statement*, (ii) a *block*, (iii) a *method*, (iv) an *object*, (v) a *class*, or (vi) an *exception*. Statement (or block) handlers are attached to a statement (or a statement block), allowing context-dependent responses to an exception. The block is usually defined by means of keywords of the language; the protected region starts with a specific keyword and ends with another keyword of the language. Method handlers are associated with a method; when an exception is raised in one of the statements of the method, the method handler attached to this exception is executed. Exception mechanisms that allow to attach handlers to blocks, consequently also support method handlers since a block may be defined as a method. Object handlers are valid for particular instances of a class; that is, each instance has its own set of handlers. Object handlers are usually attached to object variables in their declarations. Class handlers are attached to classes, allowing the software developers to define a common exceptional behavior for a class of objects. Handlers attached to exceptions themselves are always invoked if no more specific handlers can be found. They are the most general handlers, and must be valid in any part of the program, independent of execution context and object state. For instance, such a handler could print an error message or make a general correction action.

Handler Binding. When an exception is raised at run-time, a handler should be invoked to deal with the exception occurrence. There are three different design solutions for binding handlers to exception occurrences: (i) the *static* approach, (ii) the *dynamic* approach, and (iii) the *semi-dynamic* approach. In the static approach, a handler is statically attached to a protected region, and this handler is used for all occurrences of the corresponding exception during the execution of that protected region. The handler binding is independent of the control flow of the program, and hence there is no run-time search to bind handlers to exception occurrences.

In the dynamic approach, the binding depends upon the control flow of the program. As a consequence, this approach determines at run-time which handler should be used for a given exception occurrence. The handler cannot be determined at compile-time. Generally, exception handlers are defined dynamically in the executable statements of programs by executing a statement making a handler available for a particular exception. In PL/I, the binding is performed dynamically by means of the statement ON. A statement ON specifies a handler binding to a specific exception, and it stays in effect until either a new statement ON for that exception is executed or the block in which it occurs is exited.

The semi-dynamic binding is a hybrid model that combines the two previous approaches. Local handlers can be bound statically to the signaler. If a handler is not attached to the raised exception in the context of the signaler, a dynamic approach is employed to find a suitable handler. Firstly, handlers attached to enclosing protected regions are searched dynamically. If none is found, the exception mechanism then signals the exception to the

caller. The call chain of method invocations and protected regions is therefore traversed backwards until a statement or another protected region is found in which a handler for that exception is attached. The example in the Figure 2.2 illustrates the semi-dynamic approach. The exception mechanism does not find a local handler attached statically to the signaler, the method m2. The exception mechanism then proceeds the search by taking the invocation history into account. A handler is then found at the context of m1.

Exception Propagation. If no local handler is defined for an exception which has been raised, the exception can be propagated to the caller of the method raising the exception. In fact, the caller often knows what effect the operation had to achieve and how best to respond to exceptions [13, 64]. If no handler is found for the exception within the caller, the exception can be propagated to higher-level components other than its immediate caller. There are two design solutions for exception propagation: (i) *explicit* propagation, and (ii) *automatic* (or *implicit*) propagation. In the first case, the handling of signaled exceptions is limited to the immediate caller; however, the raised exception or a new exception can be signaled explicitly within a handler (attached to the caller) to a higher-level component. For this reason, the exception mechanisms that adopt such an approach are called *single-level* [40]. If a signaled exception is not handled at the caller's context, either a predefined general exception is further propagated automatically, or the program is terminated.

In the second 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. As a consequence, the exception mechanisms that incorporate this design solution are termed *multi-level* [40].

Exception propagation is closely related to the issue of handler binding. Exception mechanisms that implement static binding cannot allow exception propagation since the binding is done at compile time and the chain of invokers is ignored. Semi-dynamic and dynamic bindings are performed at run-time and, therefore, can allow exception propagation.

Continuation of the Control Flow. After an exception is raised and the corresponding handler is executed, the system should continue its normal execution. There is an issue concerning the semantics which determines the continuation of the control flow, i.e., where normal execution proceeds. There are at least two possible design solutions, which correspond to different styles of continuation of the control flow: (i) the *resumption* model, and (ii) the *termination* model. In the resumption model, the execution has the capability to resume the internal activity of the signaler after the point at which the ex-

ception was raised. In the termination model, the activity of the component which raised the exception cannot be resumed; conceptually, this means that its activity is terminated.

There are some variations of the termination model. Such variations can be classified into at least three different ways with their respective semantics: (i) *return* – terminate the signaler and direct control flow to the statement following the protected region where the exception was handled; (ii) *strict termination* – terminate the program and return control to the operating environment; and (iii) *retry the signaler* – terminate the signaler and retry it to attempt to complete the required service in the normal manner.

Figure 2.2 pictures an example of termination with return semantics. The execution of m2 (the signaler) is terminated, the handler is executed, and the normal control flow is directed to the statement following the protected region where the exception is handled. Then, execution continues at the method main since the method m1 is the protected region where the exception is handled. If the specified model of continuation of control flow was resumption, the handler would be executed and normal execution would resume the internal activity of the signaler after the point at which the exception was raised. This return point is indicated as * in Figure 2.2.

Cleanup Actions. Components of a program should be kept in a consistent state, regardless of whether the code completes normally or is interrupted by an exception. In this sense, it is required to do some cleanup action to keep the program in a consistent state before the termination of the component. Cleanup code may either restore the component to a possible state, undo some undesirable effect, or release allocated resources. Cleanup actions can be supported by particular designs of exception mechanisms in three different ways: (i) *use of explicit propagation*, (ii) *specific construct*, and (iii) *automatic cleanup*. In the first approach, the explicit propagation is used for performing some cleanup actions before termination of the component. When an exception occurrence is detected, if it cannot be handled by the signaler and has to be propagated, then the cleanup action should be specified within the corresponding handler before the statement that propagates the exception.

The second method provides a construct which is executed whenever the protected program unit exits. The cleanup code is attached to the protected program unit and this code is executed whether an exception is raised or not. If no exceptions are raised in the protected region, the attached cleanup code is executed after the protected region. However, if an exception is raised in the protected region, control is transferred immediately to the statements devoted to clean up.

The third solution is based on the premise that the exception mechanism knows what should be cleaned up before termination of the component. The exception mechanism itself performs automatically the necessary cleanup actions and the application developer

does not need to worry about.

Reliability Checks. Reliability checks test for possible errors introduced by the use of an exception mechanism. A number of issues can be checked by the exception mechanism itself, such as [73]: (i) checking that each exception is signaled with the correct set of actual parameters; (ii) checking that each handler for an exception is defined with the correct set of formal parameters; (iii) checking that only those exceptions that are defined by a signaler are signaled by it, in effect forcing the explicit propagation; and (iv) checking that all exceptions that can be raised in a given scope are handled in that scope. We classify the design approaches regarding reliability checks into two design solutions: (i) *static checks*, and (ii) *dynamic checks*. Static checks are performed by the compiler while dynamic checks are performed by the run-time system. Static checking depends on the use of exception interface, static binding and representation of exceptions as objects. When exceptions are not declared in the external interface of their signalers or are not typed, there is very little that can be checked at compile time. Some exception mechanisms do not provide any support for static checks, while other ones perform both static and dynamic checks.

Concurrent Exception Handling. When concurrent programming is supported by the underlying programming language, one or more exceptions can be raised concurrently during a cooperative activity (Section 2.2.2). In this way, exception mechanisms should provide some support for concurrent exception handling. The design approaches to concurrent exception handling can be classified into at least three possible design solutions, which correspond to different support levels: (i) *unsupported*, (ii) *limited*, and (iii) *complete*. In the first approach, no support for concurrent exception handling is provided.

Exception mechanisms that implement the second approach only provide basic support for concurrent exception handling. A special exception (signal) is used to notify the threads involved in a cooperation when an exception is raised in one of the cooperating threads. In this way, exceptions can be handled by more than one thread in a coordinated manner. Effective facilities which allow using atomic actions with exception handling are not provided. In addition, the exception resolution process is also left to application programmers.

The third approach provides complete support for concurrent exception handling. Explicit facilities are provided to use atomic actions with concurrent exception handling. Software designers can concentrate on definition of cooperative activities, exceptions and handlers, which are application-dependent issues. The exception mechanism provides: (i) synchronization of the action participants, (ii) support for exception resolution, and (iii) invocation of the different handlers attached to the action participants.

2.4 Exception Handling in Various OO Languages

In the 1970s, exception mechanisms were developed specifically for procedural programming languages. PL/I [41] pioneered the concept of providing application programmers with linguistic constructs for exception handling. However, such constructs resulted in an exception mechanism that was complex and difficult to use. The CLU's exception mechanism [40] overcame some difficulties detected in the PL/I's exception mechanism and introduced an exception handling model more suitable for implementing dependable software. In the 1980s, the object-oriented languages brought to developers a new way of thinking about and designing their systems as well as some new techniques to make them more modular and reusable. Exception handling mechanisms have been integrated into main stream object-oriented languages such as Java [30], Modula-3 [50] and Eiffel [44]. We can now review various exception mechanisms dedicated to object-oriented languages. We use the taxonomy described in Section 2.3 to help compare and evaluate their main strengths and weaknesses. Figure 2.4 in Section 2.5 summarizes the design choices of each exception mechanism presented in this Section.

CLU [40] was the first language to offer an exception mechanism more suitable for implementing fault-tolerant software. In fact, its primary purpose is to support construction of software modules which are able to respond reasonably to wide variety of circumstances. In addition, the CLU's exception mechanism is based on a simple model of exception handling that is to lead to well-structured programs. As a consequence, it is considered to be a baseline of our study.

The Exception Handling Model of CLU. In the CLU's exception handling model, exceptions are represented as symbols. However, a set of typed parameters can be used to pass information about the exception from the signaler to the handler. CLU's exception mechanism is said to be single level since the exception raised by a procedure is normally handled by its immediate caller. However, the immediate caller may resignal the exception explicitly to its invoker. A CLU procedure definition must include exception interface. Handlers are attached to any statement in a CLU procedure by clauses **except** having the following syntactic form:

```
...
statement except when E1: ... when E2: ...
...
```

The handlers are often collected at the end of the procedure whenever possible because the placement of handlers in individual statement can reduce the code readability. Howe-

ver, there is the possibility of interleaving exception handler and normal code on a per statement basis. If a statement calls a procedure that signals an exception, but that statement has no attached handler for that exception, then the exception is propagated automatically to progressively larger static scopes within the procedure. If a handler is found in the procedure, it is executed. Otherwise, the predefined default exception failure is raised and control returns to the caller. This exception is the only one implicitly propagated. CLU procedures that raise an exception are normally terminated. After the handler execution, control simply directs to the statement following the statement to which the handler is attached, that is, the termination model has a return semantics. The model does not provide any specific construct to define cleanup actions; and also does not support concurrent exception handling.

2.4.1 Exception Handling in Ada95

Ada95 [66] is a fully object-oriented language which has the upward compatibility with Ada. The exception mechanism of Ada95 is basically the same as it was in Ada; it was not revised as it could have been to become more object-oriented. For instance, Ada exceptions are originally symbols and are not declared in the external interface of the procedures (methods). However, some new features make using exceptions much simpler: (i) an exception can be raised with a message which can be analyzed during handling; (ii) the value of the variable of the new type ExceptionOccurrence represents each occurrence of the exception – there is a function converting the variable of this type into a string; (iii) another function, of the type string, returns the name of the raised exception.

Handlers can be attached to blocks, procedures or packages. Handlers are placed together in an clause exception, which must be placed at the end of the protected region, as below:

```
begin
... -- protected region -- ...
exception when E1 => ... -- handler -- ...
          when E2 => ... -- handler -- ...
          when others => ... -- all-encompassing handler -- ...
end;
```

Ada allows the definition of all-encompassing default handlers by means of the construct when others. This handler catches just those exceptions that the programmer has not provided specific handlers for.

Ada95's exception handling model supports semi-dynamic binding and automatic propagation of all unhandled exceptions. Therefore, Ada95's exception mechanism is multi-level as opposed to the single-level mechanism of CLU. It adopts the termination model with return semantics. An exception can also be propagated explicitly by a component reraising the exception within the handler. Therefore, explicit propagation can be used to perform any final cleanup actions before signaling the exception. All-encompassing default handlers can be also used to do it. However, no explicit support is provided for cleanup actions. As exceptions are not typed and are not declared in the external interface of their signalers, there is very little that can be checked at compile time.

Ada95 allows to attach handlers to tasks (threads). Handlers may be called in several concurrent tasks when an exception has been raised in one of them. However, this language has a limited form of concurrent-specific exception propagation: an exception is propagated to both the caller and callee tasks if it is raised during the Ada rendezvous. Then the exception mechanism is not applicable directly for systems that contain complex cooperative concurrency.

2.4.2 Exception Handling in Lore

An object-oriented exception mechanism has been designed by Dony [15, 16] and it has been implemented in Lore, an object-oriented language dedicated to knowledge representation. An important characteristic of this mechanism is the object-oriented representation of exceptions: exceptions are full objects. The exception interface may be part of a method's signature by means of the clause signals.

The exception mechanism ensures explicit separation between normal and exceptional code since handlers are ordinary methods of a specific class named protected-handler. Handlers can be attached to statements, classes and exception classes. Handlers attached to classes are called default handlers. When an exception is raised, the handler search proceeds as follows: (i) first handlers that are attached to statements that dynamically include the signaling one are searched, and the search stops as soon as a handler, whose parameter type is a supertype of the signaled exception, is found; (ii) if none is found, the system tries to find default handlers attached to the class or upper classes of the signaling active object; (iii) if none is found, the system looks for default handlers attached to the signaled exception itself; (iv) if none is found, the exception is then propagated automatically to the operation caller, and the sequence of steps is again repeated. Exceptions can also be propagated explicitly.

This exception mechanism is more flexible than CLU's one. Two policies for continuation of control flow are provided: resumption and termination (with return semantics). The behavior of these policies are defined as methods on an exception class from which

all exception classes inherit. The methods that define such policies are invoked within handlers. When handlers do not explicitly choose one of these options, the predefined exception ExceptionNotHandled is signaled. This approach is interesting because it tries to integrate exceptions into the standard invocation mechanism. The Lore's exception mechanism has explicit support for specifying cleanup actions. It provides the construct when-exit allowing to attach cleanup actions to expressions. No support is provided for concurrent programming and concurrent exception handling.

2.4.3 Exception Handling in Smalltalk-80

Being one of the earliest widely available object-oriented languages, Smalltalk [28] attracted much attention during the 1980s. In Smalltalk-80, an exception is a selector but not a first-class object. A selector specifies the operation name. Thus a exception selector cannot own any characteristics, cannot be inspected, modified or upgraded [16]. In order to signal run-time exceptions, the Smalltalk evaluator sends, to the current object, a message corresponding to the current exception. Therefore handlers are methods pointed out by exception selectors, and they only can be attached to classes. Thus, exceptions raised by methods defined on a class are handled within that class. Exceptions cannot be propagated to operation callers. For this reason, exception interface is not part of a method's signature.

Smalltalk-80 has no static type checking. As long as method has no syntax errors and no undeclared variables it will compile, and if there are any type errors they will occur at run-time. As a consequence, in the Smalltalk environment, most run-time errors (exceptions) occurs when a message for which no method exists arrives at an object. In this case, the run-time system sends a special message doesNotUnderstand: to the receiver, with the message selector and arguments of the original message as arguments. The search for the selector doesNotUnderstand: thus follows the same path as the search for the first selector. In this way, a user is given the opportunity to define methods doesNotUnderstand: in the visited classes, in order to customize exception handling or catch errors. If the programmer does not do this, an error is signaled anyway by the standard method doesNotUnderstand: on the class Object (all application classes are derived from Object); it causes a notifier to appear on the screen giving some information about the error, and providing the ability to invoke the debugger. Unlike CLU, Smalltalk supports both resumption and termination. Two variants of the termination model are supported: return and retry. The programmer may specify cleanup actions within an block ensure which is always executed after the protected block, no matter if an exception occurred or not. The mechanism does not support concurrent programming and concurrent exception handling.

2.4.4 Exception Handling in Eiffel

The exception mechanism of the sequential language Eiffel [44, 45, 67] is integrated with the notion of design by contract. Classes and methods establish contracts with their clients by specifying assertions: pre and post-conditions, and invariants. Exceptions are defined as the violation of assertions during the execution of the associated method, and they are raised implicitly. However, exceptions may also be defined by the user. Eiffel exceptions are typed entities which have an integer value and a string tag. However, exception interface is unsupported. Handlers can be attached to a method or a class. Thus, on the one hand the exception mechanism provides no support for attaching handlers to units at lower levels, such as a block of statements, but on the other hand it ensures explicit separation between normal and handler code.

When an exception occurs during the execution of a method, its execution is stopped and the respective handler is executed. Within a handler, the exception can be determined by comparing a predefined variable called exception with an exception name. The variables which are visible to the protected region have the same visibility in the handler. If no handler is defined, the method is said to fail and the exception is propagated implicitly to the caller (the so-called *organized panic*). Therefore, Eiffel design adopts automatic propagation as default behavior. However, exceptions can be propagated explicitly. Handlers can report failure to the caller by reraising the exception. Explicit propagation can be used for performing some cleanup actions before reraising the exception.

The exception mechanism of Eiffel supports retry, a variation of the termination model. Handlers try to restore the class invariant by retrying the method if the pre-condition still holds. In this way, a routine may succeed or fail, there is no intermediate ground. So the raising of an exception means the failure of a software component which has been unable to terminate in a normal manner. No support is provided for concurrent programming and concurrent exception handling.

2.4.5 Exception Handling in Modula-3

Modula-3's exception mechanism [50] is based on a semantic model similar to Ada95. Exceptions are represented as symbols. However, exceptions optionally can have parameters. The exception interface may be part of a method's signature by means of the clause set. As a consequence, this feature facilitates static checks for raising unlisted exceptions within the code and dynamic checks for exception occurrences that were not explicitly raised. Handlers can be attached to a statement or a block. Like Ada95, handlers are attached to a block of instructions by means of the construct try... except, as in:

```
try ... -- protected region -- ...
except ... -- handlers -- ...
else ... -- all-encompassing handler -- ...
```

Handler binding is semi-dynamic. If, during the execution of a block try (the protected region), an exception occurrence is detected, execution ceases, and control passes to the corresponding handler. If no handler is found, and the part else is present, the control flow is deviated to this part. The part else implements an all-encompassing handler, i.e. a single default handler attached to the protected region to handle any exceptions that the programmer have not provided specific handlers for. It is similar to the construct when others of Ada95. If the part else also is not present, a handler is sought in the statically enclosing protected region (a construct try... except may be nested in a block try). If no handler is found there, the exception is automatically propagated, and the search continues in the context of the calling procedure. If no handler can be found, the Modula-3 run-time system will halt the program with a suitable error message. Thus, automatic propagation is adopted as default behavior. However, exceptions also may be propagated explicitly.

When a handler is found and it has finished its execution, control passes to the statement following the protected region where the exception was handled. That is, the termination model is adopted with the return semantics (as CLU). Modula-3 provides programmers with the construct try... finally to define cleanup actions. No support is provided for concurrent exception handling.

2.4.6 Exception Handling in C++

In C++ [35], exceptions are data objects and exception interface may be optionally included as part of a method's signature. In other words, although exception interface is supported, it is not obliged. Like Ada95 and Modula-3, C++ also introduces an exception mechanism that is sensitive to contexts. The handling context is termed a block try, and handlers may be attached to a statement or a block. Handlers are declared at the end of a block try using the keyword catch. The exception is handled by invoking an appropriate handler selected from a list of handlers found immediately after the block try, as in:

```
try { ... -- protected region -- ...
} catch (E1)  { ... -- handler -- ... }
  catch (E2)  { ... -- handler -- ... }
  catch (...) { ... -- all-encompassing handler -- ... }
```

A handler catches an exception object by specifying its type. The handler declares its parameter as being of a given class, but may catch exception objects of any subclass. C++ also allows the definition of all-encompassing default handlers by means of the construct catch (...). As in Ada 95 and Modula-3, this handler catches those exceptions that the programmer have not provided specific handlers for.

Unlike CLU, the exception mechanism of C++ implements semi-dynamic binding and supports both automatic and explicit propagation of exceptions. However, the default behavior is automatic propagation. Regarding continuation of the control flow, only the termination model with return semantics is implemented. If a clause catch terminates without raising another exception, execution continues normally at the first statement after the block try to which the executed handler is attached. The model does not provide any specific support for cleanup actions. As exceptions are typed entities, static checks can be performed by the compiler. However, as the use of exception interface is not forced, dynamic checks are performed by the run-time system. No support is provided for concurrent exception handling.

2.4.7 Exception Handling in Java

Java is considered to be a language from the C++ family and adopts various similar design solutions. For instance, Java supports representation of exceptions as data objects, semi-dynamic binding, and the termination model with return semantics. In addition, Java [30, 52] provides software developers with a block try to define protected regions. However, its exception handling is much safer and clearer than that of its ancestor. As for the main aspects of exception handling, Java has more powerful features than C++, because it allows better static checking and provides specific support for programming cleanup actions.

Java adopts a hybrid solution for exception interface. All exceptions must be throwable, that is, exceptions must inherit (directly or indirectly) from the class Throwable. Classes Throwable can be categorized into two groups: (i) classes that inherit from the class Error or that inherit from the class RuntimeException are *unchecked*, and (ii) other classes that inherit from the class Exception are *checked*. The first group are exceptions from which ordinary programs are not expected to recover (for example, loading and linkage errors, virtual machine errors), or exceptions that occur within the Java run-time system (for example, arithmetic, pointer, indexing exceptions). The compiler does not require that programmers check and specify the unchecked exceptions as part of a method's signature. But Java requires that the program either catch or specify all checked exceptions that can be thrown directly or indirectly within the scope of the method.

As opposed to C++, Java provides programmers with the construct try... finally to

define cleanup actions. The block finally is always executed at the end of the block try, whether an exception is raised or not, unless the block try raises an exception that is not caught by its handlers, in which case the exception is propagated.

Although Java describes clear semantics of exception handling in concurrent Java programs, it does not offer complete support for concurrent exception handling. The Java's exception mechanism is integrated with the Java thread/synchronization model, so that locks are released as statements synchronized and invocations of methods synchronized complete abruptly. An asynchronous exception (signal) can be raised in a concurrent program by invoking the method stop on the class Thread.

2.4.8 Exception Handling in Object Pascal/Delphi

Object Pascal [4, 5] is the underlying programming language of Delphi, a tool for rapid application development. Exceptions in Object Pascal are data objects, and exception interface is unsupported. Like Modula-3 and Java, handlers can be attached to a statement or a block. A protected region starts with the keyword try and ends with the keyword end. As Modula-3 and Ada95, Object Pascal also allows the definition of all-encompassing default handlers. After exception is handled, execution continues at the end of the current block where the exception was handled. Therefore, Object Pascal implements the termination model with return semantics.

Unlike CLU, exceptions are propagated automatically and handlers may be associated semi-dynamically. If a block does not handle a particular exception, execution leaves that block and returns to the block that contains the block (or to the code that called the block), with the exception still raised. This process repeats with increasingly broad scope until either execution reaches the outermost scope of the application or a block at some level handles the exception. However, a handler may reraise explicitly the same exception by calling the keyword raise without the exception object argument. Like Modula-3 and Java, cleanup actions may be specified by using the construct try... finally. The application always executes any statements in the block finally, even if an exception occurs in the protected block. As semi-dynamic binding is adopted and exception interface is unsupported, there is very little that can be checked statically. Moreover, no support is provided for concurrent programming and concurrent exception handling.

2.4.9 Exception Handling in Guide

Guide's exception mechanism [1, 36] is similar to CLU's one. The nature of a Guide exception is symbol. However, complementary information can be passed to handlers while raising exceptions. Guide implicitly provides the handler with the names of the class and the method that signaled the exception. In Guide, exceptions potentially raised

must be included in the interface. In other words, the use of exception interface is obliged, and it is not optional like in C++ and Modula-3. Handlers may be attached to method invocations (statements), methods and classes.

If a statement raises an exception, the method containing the signaling statement signals the exception to its caller. Local handlers are not possible. Guide's exception mechanism adopts the termination model with return semantics. The normal continuation after the execution of a handler is from the point just after the raising method invocation. The binding is semi-dynamic, and all exceptions propagated from a method to its invoker should be either explicitly propagated or resignaled. Exceptions are not propagated automatically in Guide. When an exception is not handled by the caller, a special exception termed Uncaught_Exception is propagated. The retry policy also is provided through the keyword retry that only a handler can use.

Guide's approach addresses the issue of consistency of objects. The block restore allows a block with cleanup actions to be defined. The block is executed just after the raising of the exception and prior to the execution of the handler. Recursively, if the handler propagates an exception, then the block restore of the caller object is executed before the search for a new handler. As Guide uses static binding and the use of exception interface is obliged, most checks are performed statically. Although Guide provides constructs for concurrent programming, its exception mechanism does not support concurrent exception handling.

2.4.10 Exception Handling in Extended Ada

Cui's approach, called data-oriented exception handling [13], is a design that associates handlers with objects in their declarations. This concept has been implemented with an Ada preprocessor and empirical studies [13] have shown that its use can produce programs that are smaller and better structured when compared to the programs produced using Ada's traditional exception handling. In Ada's exception handling mechanism, although handlers appear after the main algorithm, introducing blocks in the middle of a statement list to associate different handlers with different objects inserts exception handling code in the middle of the main algorithm preventing a clear separation between them. The data-oriented exception handling removes exception handling code from algorithmic code helping code writeability and readability.

Exceptions are declared with type declarations in generic package specifications (exception interface). Handlers are attached statically to object variables in declarations. Each object declared has its own set of (exception, handler) binding pairs specified in its declaration. Three language features are defined to implement this design: #exception, #when, and #raise. Exceptions are declared by attaching a clause #exception to the type

exported from the specification part of a package. Handlers are associated with data object's declaration by attaching a clause #when to the declaration that specifies handler procedures for the exceptions defined on the object's type. Exceptions are signaled by statements #raise that transmit parameters, indicating the object with failure. Default handlers for exceptions can be specified in a type declaration and inherited by variables declared with that type. Like CLU, only the termination model is supported.

2.4.11 Exception Handling in Beta

Beta [42] is an object-oriented language that generalizes some of the concepts introduced by Simula67. Beta has no special constructs for exception handling. Instead a language construct, using existing syntax, is adopted. Beta's abstraction mechanism is called *pattern* which replaces classes, procedures, functions, types and exceptions. Instances of patterns are called objects and can be used as variables, data structures, procedures, functions, and so on. Inheritance is implemented using the supperpattern mechanism. This includes explicit control of overriding using the construct inner. Exceptions are represented as *virtual patterns*, a variant of the pattern describing the construction of classes or individual objects.

Signaling an exception amounts to directly calling the handler by its name. Handler can be attached to classes, objects, methods and statements. The mechanism is based on the static binding approach, i.e., there is no run-time search to find handlers. The default continuation of control flow is strict termination of the program. Beta also allows resumption.

Propagation of an exception to the caller is not supported. For this reason, the concept of searching dynamically for a handler does not exist in Beta. The construct inner and virtual patterns of Beta together provide a way for an invoker of an operation to affect the handling of an exception inside the object. The code extension (called binding) given by the invoker at the time of an invocation is executed by substituting in the operation's code at the place where inner is declared. The mechanism inner also allows a subclass to extend or augment the exception handling of the parent class. This use of inner can often require a careful understanding of the pattern's code [46].

2.4.12 Exception Handling in Arche

Arche [32, 33] is a concurrent object-oriented language which makes a clear distinction between type (description of an interface) and class (an implementation of a type). Exceptions are data objects and exception interface is supported. The clause signals may be used in any operation signature to state the exceptions that the operation may signal.

Handlers can be attached to blocks and are declared by means of a construct similar to Modula-3. Like CLU, Arche adopts the termination model with explicit propagation of exceptions. A handler may propagate the handled exception by using the command signal. Thus, the search for a handler is implemented according to the explicit propagation of exceptions. If the search fails the predefined exception failure is signaled. The handler binding is semi-dynamic. However the absence of handlers can be detected at compile time due to explicit propagation, allowing an error report to the programmer.

Of the reviewed object-oriented languages in this study, Arche provides the best scheme for concurrent exception handling. Cooperating threads can be enclosed within a scheme of object groups. Object groups are declared as a sequence through the use of the type constructor seq of. Methods of an object group are executed concurrently and synchronized within group scope and are called *multi-operations*. In Arche, the notion of multi-operations is therefore the base structuring mechanism for fault tolerance. Multi-operations can be regarded as atomic actions (Section 2.2.2).

A multi-operation may issue a coordinated call, a natural extension of the method-call mechanism. All the group components then join together to call a multi-operation and are all synchronized. When the call is terminated, results - if any - are made available to all callers' components before their parallel activities are resumed. In a multi-operation execution, many components may concurrently signal different exceptions. Then Arche provides a resolution function, which is declared within a class. A resolution function takes a sequence of exceptions as input parameter and returns an exception called *concerted exception*. The resolution function is then implicitly invoked when the execution of a multi-operation results in the signal of an exception by at least one of the multi-operation components.

2.4.13 Exception Handling in Other Languages

Trellis/Owl [18] is an important landmark in the history of object-oriented language design. When an operation is invoked and it is unable to complete, the interface of the operation has a list of exceptions that the operation can signal. The error message not found (a familiar run-time error when programming in an untyped language like Smalltalk) could not occur in the strict compile-time type checking of Trellis/Owl. Exceptions are created with the keyword signal, and handlers are defined with the clause except on, similar to CLU's exception handling.

Act1 [39] is an object-oriented language designed on the basis of the actor model. Each actor delegates the message to which it cannot respond, to another actor called its proxy. The proxy takes charge of the task it has been delegated and the delegating actor is ready to process another message. The proxy knows the delegating actor in case it should need

additional information. Delegation, like class inheritance, is a means of sharing behavior (i.e. operations). However, delegation is more flexible than inheritance: an actor may dynamically choose new proxies, whereas the inheritance graph is statically defined at compile time. A specific actor called Object is the universal proxy and the root of the "delegation tree". In this context, error handling is distributed: the continuation of a message may be one of the actors which handle exceptions, by activating an interactive debugger. Users can thus define new actors processing exceptions in a way suited to their requirements.

2.5 Evaluation and Discussion

Although it is difficult to claim which model and mechanism implemented for a single language is better than the others, the relative advantages and disadvantages of each mechanism will be identified and addressed below. Figure 2.4 provides a summary of the main aspects of the exception mechanisms presented in Section 2.4. The figure highlights the different approaches for each design issue identified according our taxonomy (Section 2.3).

Exception Representation. With respect to exception representation, we can observe that 6 mechanisms have represented exceptions as symbols. The others have represented exceptions as objects: 4 mechanisms have represented exceptions as data objects, and only 2 have chosen to represent exceptions as full objects. We can conclude that several designs still intermingle object-oriented solutions with conventional solutions since half the studied exception mechanisms have represented exceptions as symbols. From the perspective of language uniformity, notions related to exception handling should be defined according to the object-oriented programming paradigm [32, 33]. In fact, object paradigm was born during a time when procedural programming was abundantly dominant. As a result, some mechanisms of languages, called hybrid languages, combine procedural solutions of programming with object-oriented ones.

The handlers of a dependable system need useful information to damage assessment and consequent error handling. Handlers may not be in the same context where the design fault which caused the exception raising. Extra-information should be passed to the corresponding handler so that it can perform correct and effective error handling. When an exception is a mere symbol, it cannot pass parameters back to its handler. This forces the programmer to communicate via global variables, which in turn decreases the modularity of the system. Extra-information passing can be performed naturally when exceptions are represented as objects. The representation of exceptions as objects allows the inclusion of context-related information that can be passed implicitly by the exception

Taxonomy	Exception Mechanisms	Ada95	Lore	Smalltalk	Eiffel	Modula3	C++	Java	Delphi	Guide	Ext. Ada	Beta	Arche
Exception Representation	Symbols	✓		✓	✓	✓				✓	✓		
	Data Objects						✓	✓	✓				✓
	Full Objects		✓									✓	
Exception Interface	Unsupported	✓		✓	✓				✓			✓	
	Optional		✓			✓	✓						✓
	Obliged									✓	✓		
	Hybrid							✓					
Handler Attachment	Statement	✓	✓			✓	✓	✓	✓	✓		✓	✓
	Block	✓				✓	✓	✓	✓				✓
	Method	✓			✓	✓	✓	✓	✓	✓		✓	✓
	Object										✓	✓	
	Class	✓	✓	✓	✓					✓	✓	✓	
	Exception		✓									✓	
Handler Binding	Static			✓							✓	✓	
	Dynamic												
	Semi-Dynamic	✓	✓		✓	✓	✓	✓	✓	✓			✓
Exception Propagation	Automatic	✓			✓	✓	✓	✓	✓				
	Explicit	✓	✓		✓	✓	✓	✓	✓	✓			✓
Continuation of the Control Flow	Resumption		✓	✓								✓	
	Termination	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Cleanup Actions	Unsupported										✓		
	Use of Explicit Propagation	✓			✓		✓						✓
	Specific Construct		✓	✓		✓		✓	✓	✓		✓	
	Automatic Cleanup												
Reliability Checks	Dynamic Checks	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
	Static Checks				✓	✓	✓	✓	✓	✓	✓	✓	✓
Concurrent Exception Handling	Unsupported		✓	✓	✓	✓	✓		✓	✓	✓	✓	
	Limited	✓						✓					
	Complete												✓

Figura 2.4: Summary of the Features of the Exception Mechanisms

mechanism at run-time as well as explicitly by the signaler declaring the information in the state of the exception object.

Exception Interface. As regards the exception interface, 5 exception mechanisms have supported it, 4 mechanisms have not forced its use, 2 mechanisms have forced its use, and only 1 mechanism has adopted the hybrid solution. Java is the only design that adopts the hybrid solution. Designers of exception mechanisms have opted for a flexible solution regarding exception interface; programmers usually are not required to specify predefined and user-defined exceptions that a method may signal. However, the inclusion of exception interface in the method's signature leads to better readability [8]. This feature allows a programmer to state the intent of a method in a precise way, by specifying both its expected normal and exceptional behaviors [27]. Knowing which exceptions a called method may signal, the client code may guard easily against exceptional behaviors by providing appropriate handlers [27]. This is in line with the idealized-fault-tolerant-component model, i.e., components with well-defined interfaces which involves constraining the patterns of normal and exceptional interactions among the components.

Handler Attachment. With respect to handler attachment, 9 mechanisms have included statement handlers, 6 mechanisms have included block handlers, only 2 mechanisms (Extended Ada and Beta) have included handlers at the level of objects, 7 mechanisms have included class handlers, and only 2 mechanisms (Lore and Beta) have included exception handlers. For the purpose of improving the writeability and structuring of the dependable systems, it is desirable to allow multi-level attachment of handlers, i.e., the attachment of handlers to several levels of protected regions such as exceptions, classes, objects, methods and so on. However, only the exception mechanism of Beta supports multi-level attachment of handlers.

It has been a tendency to provide programmers with high flexibility for defining the size of protected regions. In other words, several exception mechanisms have allowed software developers to attach handlers to blocks, and they can define the extent of a protected region by means of keywords. However, the use of block handlers violates explicit separation of concerns, since the exceptional code is intermingled with normal code, albeit moved to the end of statement blocks. Another disadvantage of defining protected regions as blocks of statements is that nested blocks are usually added for the sole purpose of attaching an exception handler [37]. As a result, it leads to the development of dependable software which is difficult to read, maintain and test. In addition, block handlers are not absolutely necessary. Statement handlers enable the cause of the exception to be located more precisely, and can be used without violating the separation of concerns. The exception handling model of Guide is a typical example of design that adopts only statement handlers and achieves explicit separation between

the normal and exceptional activities.

Some languages offer another feature related to the handler attachment: the all- encompassing default handler. With this feature, programmers may provide a single handler attached to a protected region to handle any exceptions that have not been provided specific handlers for. In order to use this feature in Modula-3 (Section 2.4.5), programmers add a part else to the part except of the exception-handling block. However, the use of the all-encompassing default handlers may be error prone. The part else handles all exceptions, including those the software developer know nothing about. In general, the exceptional activity of a dependable system should handle only exceptions its programmers actually know how to handle. In other cases, it is better to execute cleanup code and leave the handling to code that has more information about the exception and knows how to handle it. Several exception mechanisms, such as Delphi, Ada95 and C++, also adopt this error-prone feature in order to provide a high degree of flexibility for handler attachment.

Handler Binding. Related to handler binding, 3 mechanisms have implemented the static approach, none has implemented the dynamic approach, and 9 mechanisms have implemented the semi-dynamic approach. Static binding leads to better readability since it is easier to verify which handler would be activated for a given exception occurrence. With dynamic and semi-dynamic binding, it is more difficult since exceptions are propagated dynamically and the binding depend effectively upon the control flow at run-time. However, exception propagation is not allowed and there is no run-time search to find handlers in the static approach (Section 2.3). It has been said that not to propagate exceptional results of a conceptual level equal to that of the operation breaks reusability [8]. The callers of an operation generally have better solutions for handling than statically bound handlers (local handlers) that are unaware of the computation history.

The dynamic and semi-dynamic approaches take the invocation history into account while finding for a handler as opposed to the static approach. In fact, the handler binding in the dynamic and semi-dynamic approaches depend effectively upon the control flow at run-time. From the perspective of dependable software, it should be possible to change the currently installed handler at run-time without shutting down or rebooting the system. Most dependable object-oriented systems are essentially critical and cannot be shut down and rebooted. Therefore, some way of dynamic binding should be supported.

Exception Propagation. With regard to the exception propagation, 6 mechanisms have supported automatic propagation and 9 mechanisms have implemented the explicit propagation. The exception mechanisms of Smalltalk, Extended Ada and Beta adopt static binding and therefore have not supported any kind of propagation. Although most mechanisms allow explicit propagation of exceptions, automatic propagation is usually

adopted as default behavior. In fact, Ada95, Eiffel, Modula-3, C++, Java and Delphi have implemented both semi-dynamic binding and automatic propagation. Thus flexibility also has influenced this design issue in the different exception handling proposals. However, automatic propagation may allow an exception occurrence be inadvertently bound to an inappropriate handler. In addition, automatic propagation of unhandled exceptions through different levels of abstraction may compromise information hiding because the exception object can reveal information about the original signaler to other than its immediate caller. It decreases modularity since it can thereby increase coupling [73]. Explicit propagation addresses this problem since the handling of an exception occurrence is limited to the immediate caller in this approach. Explicit propagation of exceptions is only forced in 4 mechanisms: Lore, Guide, Extended Ada and Arche.

Continuation of the Control Flow. Related to continuation of control flow, all mechanisms have adopted the termination model. Very few languages, such as Mesa [47] and PL/I [41] (which are not addressed here), implement exclusively the resumption model which has been considered to be too complex [38, 33]. Beta, Smalltalk and Lore provide both the termination and resumption models. Mechanisms that support resumption are very powerful and flexible, but it turns out to be difficult to use by application programmers. From the viewpoint of fault tolerance, an exception mechanism should be simple and reliable. A mechanism implementing resumption has to support a more complex pattern of interaction: the system invokes a component which in turn can invoke the system by signaling an exception [38]. Unnecessary complexity may introduce error-prone features in the design of an exception mechanism and complicate the programmer's task while developing its dependable system. In fact, as far as fault tolerance is concerned the termination model is considered to be most adequate due to its clearer semantics [40]. A formal treatment of the termination model within the framework for software fault tolerance is given by Cristian [12].

Cleanup Actions. With respect to the cleanup actions, 1 mechanism (Extended Ada) has not provided any support for cleanup actions, 4 mechanisms have only allowed to specify cleanup actions by means of explicit propagation, 7 mechanisms have provided specific constructs, and none of the exception mechanisms has provided automatic facilities. Thus most exception mechanisms provide a specific construct which is executed whenever the protected region unit exits. Ideally the exception mechanism should be responsible for performing cleanup actions automatically. It could make programmer job more simple and less error prone, and would allow to achieve a number of quality requirements, such as readability, maintainability, and simplicity. However, the feasibility of this approach is doubtful since implementing automatic cleanup may be too difficult, and investigation of alternate methods is required [37].

Reliability Checks. As regards the reliability checks, 10 mechanisms have supported dynamic checks, and 9 mechanisms have supported static checks. Most mechanisms perform static checks, followed by some level of dynamic checking. Some mechanisms have implemented either exclusively dynamic checks, or only static checks. In Smalltalk, for instance, checks are all performed dynamically. It is a untyped language, if there are any type errors they will occur at run-time. Therefore Smalltalk cannot be considered an adequate language for construction of dependable object-oriented software, although the design of its exception mechanism is object-oriented. Smalltalk is more suitable for other areas of system development. In fact, different languages simply have different goals, and are tailored to meet the needs of different communities. Unlike Smalltalk, C++, Java, and many other languages, Eiffel takes the view that error handling and fault tolerance semantics should be the central part of the language. The aim of its design criteria is allow the development of robust applications. According to the Eiffel's discipline, an exception arises only if a routine fails because of some error [27]. Eiffel contains a broad range of techniques such as pre-conditions, pos-conditions and assertions, which are complementary to the exception mechanism (Section 2.4.4). With the use of these additional techniques, in most cases there is no need for naming exceptions or for providing a raise statement. All that matters is whether a failure that would violate the object's contract occurred in a method.

Concurrent Exception Handling. Related to the issue of concurrent exception handling, only the Arche language has effectively provided complete support for it. Arche supports a mechanism based on a concurrent exception-handling model whose features enforce the construction of correct and robust programs. Arche's exception mechanism 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. Moreover, the exception resolution mechanism implemented in Arche is not based on the concept of exception tree (Section 2.2.2). Issarny *et al.* [32, 33] argues that exception trees are not indicated for parametrised exception, so Arche have introduced the concept of resolution function to determine which handler should be activated in case various exceptions are raised in a cooperating group of objects. However, to program resolution functions it seems not to be an easy task since application developers are responsible for deciding how to implement resolution functions. As a consequence, concurrent programming and concurrent exception handling in Arche is not simple and can become rather difficult to use.

To summarize our discussions, the following issues conclude this Section:

- *Several design decisions are based on too flexible and complex solutions.* In spite of

the prime aim of exception mechanisms in working as a simple and reliable scheme for developing robust programs, a number of their design decisions still are based on several flexible and complex solutions. The use of flexible and complex features may lead to the construction of dependable object-oriented software which is error prone. The use of all-encompassing default handlers is an example of flexible feature with unnecessary expressive power which may cause the introduction of additional design faults.

- *Lack of support for concurrent exception handling.* The main drawback of the current exception handling techniques is the lack of complete support to handle concurrent exceptions. Only Arche has effectively provided support for concurrent exception handling. However, as stated above, its exception mechanism has some limitations. In this way, in actual concurrent object-oriented languages, exception handling is still an evolving subject where no clear consensus exists and many open problems remain.

- *Studied exception mechanisms have not fully addressed the demanding quality requirements.* None of the investigated exception mechanisms have incorporated design decisions which are fully suitable for developing dependable object-oriented software. Designers of exception mechanisms do not pay enough attention to the demanding quality requirements, such as readability, modularity, uniformity, maintainability and reusability. In addition, the advantage of one mechanism is often the disadvantage of the other. For instance, the mechanisms of Lore, Smalltalk and Extended Ada adopt design solutions highly integrated with the object paradigm, but fail in providing an exception mechanism more restrictive and ease to use. However, it is worthwhile to highlight the design of the exception mechanism of Guide which, according to our evaluation, has reached the highest punctuation in our ranking. In addition, the exception mechanism of Eiffel is interesting since it is complemented with a broad range of techniques as discussed above. We can claim that both mechanisms have a design more suitable to produce dependable object-oriented software with demanding quality attributes, although no support for concurrent exception handling is offered for these mechanisms.

2.6 General Design Criteria

The taxonomy (Section 2.3) identified the several design issues of an exception mechanism, and classified the different solutions to design them. The design decisions on these issues should be taken according to the demanding quality requirements. Figure 2.5 pictures this scenario. Note that some quality requirements are stated by the dependable

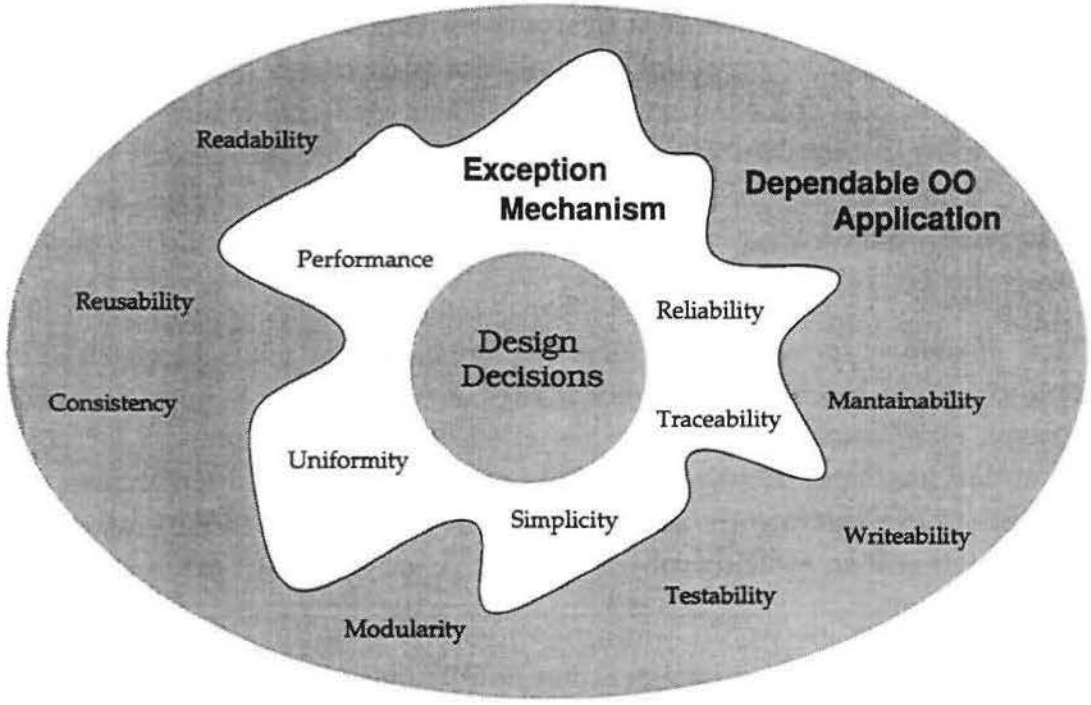


Figura 2.5: Quality Requirements of Exception Mechanisms

object-oriented applications using the exception mechanism, while others are imposed on the exception mechanism itself. However, the designs of existing exception handling mechanisms have not satisfied these requirements (Section 2.5).

This section outlines the criteria to design an effective exception mechanism for developing dependable object-oriented systems. Based on the criteria, we define the design choices for an ideal exception handling model for this kind of systems. The criteria and the proposed exception handling model have been developed based on our extensive work in building dependable object-oriented systems [19, 60, 53, 54, 63, 71, 69] and exception mechanisms for this kind of systems [23, 21, 57, 58, 61, 69].

2.6.1 Quality Requirements of an Exception Mechanism

Q1. Readability. One of the main reasons to use an exception mechanism is to promote program readability [6, 48]. The importance of readability increases regarding dependable object-oriented software since the number of possible exceptions and the exceptional activity to deal with such exceptions are both very large and complex. The exception mechanism should promote explicit separation between the exceptional and normal execution code, following the overall structure of the component-fault-tolerant-idealized model (Section 2.2.1). A mechanism with a clear separation will be easier to read and understand,

highlighting the main purpose and extent of the protected region and the abnormal code in the exception handler section. Otherwise, the code for the normal situations may then be difficult to read.

Q2. Modularity. An exception mechanism should yield dependable object-oriented applications in which the effect of an abnormal condition occurring at run-time in a component will remain confined to this component, or at least will propagate to a few neighboring component only [44]. In this way, the exception mechanism should ensure each component of a dependable object-oriented application practices information hiding [73].

Q3. Maintainability. It is widely estimated that 70% of the cost of software is devoted to maintenance [44]. An effective exception mechanism should not neglect this aspect and promote ease of program maintenance. If dependable object-oriented software can not easily changed, additional errors will be introduced during the maintenance phase. As a result, the design of an exception mechanism for dependable object-oriented systems should specially emphasize simplicity and program readability.

Q4. Reusability. Designing for reusability means that the system has been structured so that its components can be chosen from previously built products. The exceptional activity of a software component should be reused as well as the normal activity. Alternatively, exceptions and handlers are defined independent of the component, thus reused independently. If reusability is not satisfied, it compromises the ability to incorporate new exception handlers into idealized fault-tolerant components. It forces programmers to write exception-handling code even if the main body of code is already available [37].

Q5. Testability. Software testability refers to the ease with which software can be made to demonstrate its faults through (typically execution-based) testing. In general, a system's testability relates to several structural issues [2]: (i) its separation of concerns, (ii) its level of documentation, and (iii) the degree to which the system uses information hiding. Object-oriented software testing is still an evolving area. Furthermore, the addition of exceptions and exceptional behavior complicates significatively the testing activity. It should not be difficult to analytically verify that every possible error has a known handler, and it should not be hard to test every exceptional scenario in a systematic manner. Depedanble object-oriented software should be well-tested in order to decrease the possibility of manifestation of residual faults at run-time. The design decisions of an exception mechanism should be taken without damaging the testability of dependable object-oriented software.

Q6. Writeability. Dependable object-oriented systems should embed error recovery activities at various levels of the system. In this way, the complexity inherent to such systems could be controlled in a flexible and systematic approach, and redundancy could be similarly added at several levels of an object-oriented system. However, care is needed in introducing expressive power. Unnecessary expressive power may introduce additional complexity for the exception mechanism (Section 2.5).

Q7. Consistency. Components of a software system should be kept in a consistent state, regardless of whether the code completes normally or is interrupted by an exception. The consistency of components of dependable object-oriented systems should always be maintained, because such systems usually continue to execute even in the presence of errors to prevent catastrophic failures.

Q8. Reliability. The exception mechanism features should aid the development of reliable programs. Therefore, the exception mechanism should be designed to avoid error-prone features and to maximize automatic detection of programming errors. In fact, the team of designers of a dependable object-oriented software has yet to deal with many fault types. Additional faults should not be introduced by the use of the exception mechanism. The exception handling system should anticipate and prevent common programmer errors.

Q9. Simplicity. As with all language features, the exception mechanism must be simple to understand and use. Meyer [44] advocates that a good exception mechanism should be simple and modest. Therefore, the exception mechanism should not contain unnecessary complexity. It should have a consistent semantic structure that minimizes the number of underlying concepts. In other words, the concepts introduced by an exception mechanism should be as small as possible and consistent with the needs of the dependable object-oriented applications. It should have few special cases and should be composed from features that are individually simple in their semantics. In a exception mechanism that programmers of dependable systems master totally, they feel confident and can concentrate on the complexity inherent to their systems rather than on the intricacies of the exception mechanism.

Q10. Uniformity. The exception mechanism should have uniform syntactic conventions and should not provide several notations for the same concept. In addition, the design solutions of an exception mechanism for object-oriented systems should be uniformly adopted in the light of object-orientation. Object-oriented solutions should not be intermingled with conventional solutions. Otherwise, it would affect negatively reusability, modularity, testability of the dependable object-oriented software using the exception

mechanism.

Q11. Traceability. Dependable object-oriented software needs useful information to damage assessment and consequent error recovery. The information should be passed by the exception mechanism together with the notification of the exception, and it may include the name, description, location, severity of the exception, propagation chain and other useful data (Section 2.2.1).

Q12. Performance. Performance is always a consideration. There are two major trends in the exception mechanism design: (i) the time for searching a suitable handler when an exception is raised – ideally, the complexity of the search algorithm should be $O(1)$; (ii) run-time overheads caused by the exception mechanism under normal operation conditions – ideally, the mechanism should be designed so that run-time overheads are incurred only when handling an exception. However, in the case of dependable systems, in particular, where speed of error recovery is on prime importance, an application may be prepared to tolerate a little overhead on the normal error-free operation [6]. Performance frequently compromises the achievement of all other qualities. Some performance penalty should be tolerated for a greater quality of the exception mechanism. In this way, the other qualities should be priorized in designing an exception mechanism for dependable object-oriented software. Performance is really crucial in hard real-time systems which is not our study aim. We refer to [37] for deeper discussions regarding exception handling and real-time systems.

2.6.2 An Ideal Exception Handling Model

After analyzing the demanding quality requirements, we discuss each design decision and present an ideal exception handling model for dependable object-oriented software. We point out which quality requirements each design choice affects positively or negatively. Figure 2.6 summarizes these influences. Tradeoffs also are discussed since the quality requirements may conflict.

D1. Exceptions Represented as Objects. Exceptions should be represented as (full or data) objects. This design decision has a number of benefits. For instance, it leads to better traceability (Q11) and modularity (Q2) since extra-information passing can be performed naturally when exception occurrences are objects (Section 2.5). Moreover, this representation is integrated uniformly (Q10) with the object paradigm and has a number of advantages when compared to the classical approach, such as [8, 15]: (i) exceptions organized into an inheritance hierarchy which makes the system easier to reuse (Q4), read

Design Decisions		Quality Requirements	Application							Exception Mechanism				
			Q1.Readability	Q2.Modularity	Q3.Maintainability	Q4.Reusability	Q5.Testability	Q6.Writeability	Q7.Consistency	Q8.Reliability	Q9.Simplicity	Q10.Uniformity	Q11.Traceability	Q12.Performance
D1.	Exception Representation	Full Objects/Data Objects	+	+	+	+	+	+		+	+	+	+	-
		Symbols	-	-	-	-	-	-		-	-	-	-	+
D2.	Exception Interface	Unsupported/Optional	-	-	-	-	-	+		-	+	-		+
		Obliged	+	+	+	+	+	-		+	-	+		-
		Hybrid	+	+	+	+	+	-		+	-	+		-
D3.	Handler Attachment	Statement/Block	-	-	-	-	-	+			-			
		Method/Object/Class/Exception	+	+	+	+	+	-			+	+		
D4.	Handler Binding	Static	+	+		-	+			+	+			+
		Dynamic	-	-		+				-	-			-
		Semi-Dynamic	-	-		+					-			-
D5.	Exception Propagation	Automatic	-	-	-	-	-	+		-	+			+
		Explicit	+	+	+	+	+	-		+	-			-
D6.	Continuation of the Control Flow	Resumption								-	-			-
		Termination								+	+			+
D7.	Cleanup Actions	Unsupported	-						-	-				
		Use of Explicit Propagation	-					-	-	-	-			+
		Specific Construct	+					+	+		+			+
		Automatic Clenaup	+		+	+	+	+	+	+	+			-
D8.	Reliability Checks	Unsupported								-				
		Dynamic Checks								+				-
		Static Checks								+				+
D9.	Concurrent Exception Handling	Unsupported	-		-	-	-	-	-	-				
		Limited	-	-	-	-	-	-	-	-	-			-
		Complete	+	+	+	+	+	+	+	+	+			+

Figura 2.6: Design Decisions x Quality Requirements

(Q1), test (Q5), maintain (Q3) and extend; (ii) handler definition is powerful (Q6), since handlers do not only handle one kind of exception but all exceptions that are subclasses of it – consequently, less handler bindings are needed, and the program is shorter which improves readability (Q2) and makes the mechanism more simple and easier to use (Q9); (iii) handlers that are independent of any execution context can be attached to exception classes, and handlers attached to classes can be inherited by subclasses (Q6); and (iv) the use of the exception mechanism is more reliable (Q8), since representing exceptions as mere symbols may be error prone.

D2. Obliged Exception Interface. According to the idealized-fault-tolerant-component model (Section 2.2), each system component should be able to return well-defined responses. The normal and exceptional responses of the components of a dependable system should be rigorously specified. To understand which exceptional responses a method may return, one should not have to examine its implementation. In fact, the presence of exception interface leads to better readability (Q1) (Section 2.1). This feature also promotes the construction of modular software systems (Q2) [8], which in turn improves maintainability (Q3), reusability (Q4) and testability (Q5). Finally, exception interface affects the conformance rules checked by the compiler and makes the exception mechanism more reliable (Q8). Therefore, exception interface should be obliged. However, the hybrid approach also could be adopted since it is difficult to designers anticipate all exceptions, many exception types are unpredictable by nature.

D3. Multi-level Attachment of Handlers. For the purpose of improving the writeability (Q6) and structuring of dependable object-oriented software, it is desirable to allow the multi-level attachment of handlers (Section 2.5). The programmer can assume the existence of different levels of handler attachment. When an exception is related to an operation, a handler for this exception may be locally associated with the operation. Alternatively, handlers can be associated with a class, which can be applied to all operations of that class. It is also possible to attach handlers to objects and exceptions themselves. Such flexible attachment has many advantages: (i) it provides a clear separation of the object abnormal behavior from the normal one according to the concept of an idealized fault-tolerant component (Q1); (ii) protected regions can be factored out at the respective levels of classes, objects and operations (Q6); (iii) software layering facilitates the design of fault-tolerant systems; and (iv) the close integration between the language and the exception mechanism could be obtained through the uniform use of the object paradigm (Q9). However, block handlers should not be supported. The use of block handlers usually intermingles the exception handling code with the normal flow of an operation, which may result in less readable (Q1) and reusable (Q4) programs. Separation of concerns is

on prime importance for dependable object-oriented systems. The error handling code is detailed and complex and may then make code for the normal situations difficult to read (Q1) and maintain (Q3). Explicit separation of concerns achieves a number of software qualities: readability (Q1), modularity (Q2), maintainability (Q3), reusability (Q4), testability (Q5) and writeability (Q6).

D4. Semi-Dynamic Binding. As stated previously in the Section 2.5, although the static approach leads to better readability (Q1), some way of dynamic binding should be supported for dependable systems. We believe semi-dynamic binding is sufficient. Semi-dynamic binding associates different handlers with the exception in different contexts during a program's execution. In addition, the semi-dynamic binding method can be used to achieve the functionality similar to that of the dynamic method. A semi-dynamically bound handler can call different handlers based on run-time conditions. However, static binding cannot achieve this because the run-time condition may not be valid in some contexts. Although this design solution has negative influences on readability (Q1) and simplicity of the exception mechanism (Q9), the adoption of explicit propagation (D5) minimizes such negative impacts. Explicit propagation limits the handler binding to the local context related to the signaler and to the immediate caller.

D5. Explicit Propagation of Exceptions. Explicit propagation should be the only way of propagating exceptions along the chain of invokers. According to the CLU's designers, the caller of a method x should know nothing about the exceptions signaled by the methods which are called during the execution of x. The handling of an exception occurrence should be limited to the immediate caller. Explicit propagation directly improves modularity (Q2) (Section 2.5), which in turn improves readability (Q1), maintainability (Q3), reusability (Q4), testability (Q5) and reliability (Q8). The exception mechanism should therefore not provide automatic propagation and should force the users to explicitly rename any propagated exception. When automatic propagation is disallowed, the set of handlers that can field a particular exception can be statically determined, thus allowing additional compiler checks (Q8). However, this design choice obviously constrains writeability (Q6) and performance (Q12).

D6. Termination. As discussed in Section 2.5, termination should be the only supported model for continuation of the control flow. A mechanism implementing only termination is very simple to construct (and hence more reliable - Q8) since the signaling of an exception can be regarded as an abnormal return from the component [38]. From the viewpoint of fault tolerance, the resumption model introduces unnecessary expressive power [40] as well as additional complexity for the exception mechanism (Q9) [40, 33].

Practical experience with exception mechanisms providing resumption has shown that the resumption model is more error prone [27]. Furthermore, it can promote the unreliable programming practice (Q8) of removing the symptom of an error without removing the cause [27].

D7. Explicit Support for Cleanup Actions. The use of an exception mechanism might lead to inconsistencies (Q7) when exceptions are raised [64]. Components of a dependable object-oriented application cannot be left in inconsistent states, since that the system should continue to operate even in the presence of errors to prevent catastrophic failures. Automatic facilities for cleanup actions can be infeasible (Section 2.5) and it would cause probably high overheads at run-time (Q12). In fact, none of the exception mechanisms in realistic object-oriented languages have automatic cleanup. Therefore, the most suitable solution is provide programmers with specific support for cleanup actions; using explicit propagation to perform cleanup actions is more error prone (Q8), and more difficult to understand and use (Q9). Furthermore explicit support leads to better readability (Q1) and writeability (Q6) because it avoids replication of code devoted to cleanup. This problem is inevitable when using explicit propagation since cleanup actions are implemented within each handler attached to the protected region.

D8. Static Reliability Checks. The exception mechanism should be designed for creating highly reliable dependable software. It should provide extensive static checking, perhaps followed by some level of dynamic checking. This design decision is devoted to guide programmers of dependable systems towards reliable programming habits (Q8). Our decisions in adopting exception interface (D2) and explicit propagation as design principles facilitate static checking. For instance, the compiler may verify if an exception being raised at run-time will have a bound handler.

D9. Complete Support for Concurrent Exception Handling. We consider that complete support for concurrent programming as one basic aspect of an actual exception mechanism because we believe it is extremely important for realistic dependable object-oriented applications. In practice, the approach classified as limited (Section 2.2.2) can lead to production of software components which are difficult to read (Q1), maintain (Q3), reuse (Q4), and test (Q5). In addition, the responsibility related to handler invocation and exception resolution is left to application programmers which in turn leads to unreliable programming of dependable object-oriented applications (Q8). From the viewpoint of fault tolerance, concurrent exception handling is complicated (Section 2.3) and should be integrated with atomic actions. The effort of developers of dependable object-oriented

systems should be minimized and they should concentrate on issues which are application-dependent.

2.7 Ongoing Research

As we have concluded in the previous Section, existing exception mechanisms have not fully addressed an appropriate design criteria. The current lack of effective exception mechanisms for developing depedendable object-oriented software with the demanding quality attributes requires the building of new error-handling techniques. Ideally a new technique developed for a specific programming language should not introduce new language constructs. In practice this would make the approach infeasible for existing languages. We believe this is the time to map the fault tolerance approaches that are well researched but are not used in practice very often, onto practical, widely used existing languages. It seems to be one of the main flaws of the previous research specially related to fault-tolerant software that it is still rather theoretical and is applied to exotic systems and languages [59]. In addition, a new exception mechanism should be developed without conflict with other existing mechanisms.

One way to extend the facilities of programming languages is to use preprocessors which will accept an extended syntax as input and map them into the standard form of the language. Usually, such extensions however are not compatible; then other preprocessors may not be combined with each other, which results in unsolvable dilemmas [44]. A tendency for extending object-oriented programming languages is to use the computational reflection technique. This technique is based on the reflection mechanism which introduces a new dimension of modularity – the separation of the base-level computation from the meta-level computation. This approach allows to implement additional mechanisms for the underlying language without any changes to the language itself.

The work of Hof *et al.* [31] describes an exception mechanism based on meta-programming and computational reflection. Its implementation was carried out in a specific system but it could be implemented to most other systems that support meta-programming. However, such a mechanism does not support concurrent exception handling in cooperating participants and is not fully integrated with the object paradigm. The work of Garcia *et al.* [21] proposes a new error-handling technique for developing dependable object-oriented software also based on a reflective approach. The meta-level implements the exception mechanism, and at the base level resides the application. They have implemented their exception mechanism within the Java programming language without any changes to the language itself by means of a meta-object protocol. The proposed object-oriented exception handling model is based on the idealized fault-tolerant component (Section 2.2.1) and establishes a clear separation between exceptional and normal code. Mitchell *et al.* [48]

also propose an exception handling model which ensures complete separation between error handling and normal code. However, their proposal applies the reflection technique in a different way. Instead of utilizing reflective principles to achieve the separation between the application and the management mechanisms related to exception handling, this work explores reflection to separate the aplication's normal code (meta-level) from the aplication's exceptional code (base level).

Object-oriented frameworks is also an emerging technology in the world of object-orientation. A framework is a reusable and flexible software that can be extended to produce customized applications. Framework's designers specify variations within its design by means of extension points, which are those aspects of an domain that have to be kept flexible; developers of a specific application refine the framework design for the needs of their aplication by filling in those extension points. Extension points describe where and how the framework is extended and customized. We argue that framework technology is a sounding idea for implementing an exception mechanism. An exception mechanism could be implemented as an object-oriented framework providing a set of extension points since different kinds of applications would require different functionalities of the exception mechanism. For instance, dependable systems require the termination policy for the exception mechanism, but the resumption policy may be useful for simulation systems. The extension points could implement the different design approaches for each exception mechanism's functionality according to our proposed taxonomy. These extension points could be easily adapted according the context where the exception handling framework is being employed.

According to [34] the use of design patterns is extremely useful both as a guide during the framework development and as a help in better understanding a framework design. A design pattern is a microarchitecture that applies to a cross-domain design problem [7]. Some of the most useful patterns describe the framework's extension points. In this way, a system of patterns for exception handling could be developed to assist the building of an exception handling framework and document its design. The *Error Detection* pattern [56] proposes a design solution to detect errors of an application at runtime. However, such a pattern only encompasses error detection; it does not define means for the definition of handlers to cope with such exceptions. The paper [24] proposes a set of design patterns for the exception handling domain.

As we have examined in this paper, the main drawback of the current exception handling techniques is the lack of complete support to handle concurrent exceptions. Somes works have been developed to integrate concurrent exception handling with the atomic action concept. The paper [58] describes a concurrent exception mechanism based on atomic action structures for the Ada95 language. The coordinated atomic action concept (CAAction) [69] was introduced as a unified approach for structuring complex concur-

rent activities and supporting error handling between multiple interacting objects in a concurent object-oriented system. CAActions provide a suitable framework to develop dependable object-oriented systems. The paper [61] discusses the introduction of concurrent exception handling and CAAction schemes into object-oriented systems. This paper also discusses a distributed exception-resolution algorithm.

As stated previously, error handling activities play a special role in the development of dependable object-oriented software. Traditional methods of object-oriented software deal with exceptions at late design and implementation phases. Better results might be achieved if exceptions and exception handling activities might be incorporated in a consistent and disciplined way during all phases of development of a dependable object-oriented software. Instead of assuming that exception handling should be restricted to the later phases of software development, the work of de Lemos and Romanovsky [14] describes a systematic and effective approach in how to deal with exception handling at all phases of the software lifecycle. The approach provides a stepwise method for defining exceptions and their respective handlers, thus eliminating the *ad hoc* way in which exception handling is sometimes considered during the later phases of the software lifecycle.

It should there be as little extra work as possible for programmers of dependable object-oriented systems using a exception mechanism. A mechanism that provides a set of standard templates or a CASE tool to speed the implementation is often considered easier to use [37]. In addition, a number of tools could be used during all phases of the software lifecycle. Only a few researchers has dealt with this question. *Xept* [68] is a tool that can be used to add to object code the ability to detect, mask, recover and propagate exceptions from library functions. According the authors, its use helps to alleviate or avoid a large class of errors resulting from function misuses.

2.8 Concluding Remarks

Nowadays exception mechanisms are important features of object-oriented programming languages. In the context of dependable object-oriented software, exception mechanisms are used to structure the fault tolerance activities incorporated to a system. The software quality attributes of modern software systems require suitable design solutions for an exception mechanism that will be used to develop dependable object-oriented software. This paper initially presents an introduction and overview of the notions of exception handling and fault tolerance. A taxonomy for classifying the different design solutions in existing exception mechanisms has been developed. The proposed taxonomy addresses nine main aspects of interest, including exception representation, exception interface, handler attachment, handler binding, exception propagation, continuation of the control

flow, cleanup actions, reliability checks, and concurrent exception handling. The exception handling models of twelve exception mechanisms for object-oriented languages have been reviewed and evaluated with respect to the developed taxonomy. We also have defined a set of demanding quality requirements which should be satisfied while developing a proper exception mechanism for dependable object-oriented software. The defined requirements form the criteria which we have used to determine the design solutions for an ideal exception handling model. Finally, we have suggested directions for future research.

Language features and their corresponding mechanisms for exception handling continue to evolve in both experimental and commercial object-oriented languages. Our evaluation has concluded that none of the exception mechanism has addressed an appropriate design criteria. From this study, we have found that most of the existing mechanisms still adopt a number of classical design solutions for the implementation of exception handling models. In addition, several design decisions for such mechanisms are based on too flexible and complex solutions which may lead to the construction of dependable object-oriented software which is not well structured. Thus, an ideal object-oriented exception mechanism has not yet come out. This is partially because the designers of a new language does not pay enough attention to the language part that supports exception handling; in most cases, they usually attempt to add exception handling facilities to an existing language rather than to keep exception handling in mind at the very beginning of the process of language design.

However, it is worthwhile to highlight the design of the exception mechanism of Guide which, according to our evaluation, has reached the highest punctuation in our ranking. In addition, the exception mechanism of Eiffel is interesting since it is complemented with a broad range of techniques as discussed in the Section 2.5. We can claim that both mechanisms have a design more suitable to produce dependable object-oriented software with effective quality attributes, although no support for concurrent exception handling is offered for these mechanisms. In fact, the main drawback of the current exception handling techniques is the gap concerning explicit support for concurrent exception handling. Arche is the only language which has contributed a lot in this area, although it has some limitations.

2.9 Resumo do Capítulo 2

Este capítulo apresentou um artigo que aborda um estudo comparativo de mecanismos de exceções existentes em linguagens de programação orientadas a objetos. O artigo inicialmente apresenta uma revisão dos conceitos importantes relacionados a tratamento de exceções e tolerância a falhas. Uma taxonomia é proposta para classificação e comparação dos diferentes modelos de tratamento de exceções estudados. Os modelos de doze mecanismos de exceções são revisados e comparados com base na taxonomia desenvolvida. O artigo também apresenta um critério de projeto adequado para mecanismos de exceções utilizados no domínio de aplicações orientadas a objetos confiáveis. Um modelo ideal de tratamento de exceções é desenvolvido, utilizando o critério de projeto definido.

O estudo realizado neste capítulo conclui que os mecanismos de exceções estudados não incorporam um modelo de tratamento de exceções adequado para construção de software orientado a objetos confiável. Várias decisões de projeto destes mecanismos são baseadas em soluções complexas e demasiadamente flexíveis. O uso destes mecanismos pode conduzir a construção de software não confiável e que são difíceis de entender, manter e reutilizar. A principal desvantagem dos mecanismos investigados é a falta de suporte apropriado para tratamento de exceções concorrentes.

O próximo capítulo apresenta um mecanismo de exceções que adota um modelo de tratamento de exceções adequado para o contexto de aplicações orientadas a objetos confiáveis. Além disso, o modelo especialmente provê suporte para tratamento de exceções concorrentes.

Capítulo 3

Projeto e Implementação de um Mecanismo de Exceções para Software OO Confiável

O desenvolvimento de mecanismos de exceções adequados para a construção de software orientado a objetos confiável não é uma tarefa trivial. O modelo de tratamento de exceções deve prover suporte para uma separação explícita entre as atividades normais e as atividades incorporadas para tratamento de exceções de tal forma a manter sob controle a complexidade geral do sistema. O modelo deve ser integrado com o modelo de objetos e oferecer suporte para tratamento de exceções concorrentes. Um mecanismo de exceções adequado deve ser restritivo e simples de usar de tal forma que erros adicionais não sejam introduzidos pelo seu uso.

Este capítulo contém o artigo *"An Exception Handling Mechanism for Developing Dependable Object-Oriented Software Based on a Meta-Level Approach"* [21]. Este artigo foi publicado em *"Proceedings of the 10th IEEE International Symposium on Software Reliability Engineering - ISSRE'99"*, realizado em Boca Raton, Florida, Estados Unidos em novembro de 1999. O artigo apresenta o projeto e implementação de um mecanismo de tratamento de exceções para construção de software orientado a objetos confiável. O mecanismo implementa um modelo de tratamento de exceções adequado para o domínio de sistemas orientados a objetos confiáveis, contemplando o critério de projeto definido no Capítulo 2. A técnica de reflexão computacional é utilizada para implementação do mecanismo proposto.

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)
Campinas, SP – Brazil
{afgarcia, delano, cmrubira}@dcc.unicamp.br

3.1 Introduction

With software systems growing in size and complexity, the quality and cost of development and maintainence are still deep concerns for software developers. Object-oriented component-based engineering is a promising approach for 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 mechanisms 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 [36, 35]; (ii) exception handlers are represented as ordinary methods; (iii) 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 [51].

The remainder of this text is organized as follows. Section 3.2 defines the terminology adopted in this work related to exception handling and fault tolerance. Section 3.3 discusses some important design issues related to exception handling mechanisms in object-oriented languages and concurrent systems. Section 3.4 presents the concepts of computational reflection and meta-level architectures. Section 3.5 presents our exception handling model. Section 3.6 describes an example of use of the proposed mechanism. Section 3.7 describes our meta-level architecture for exception handling. Section 3.8 gives a brief comparison with related work. Finally, Section 3.9 summarizes the conclusions of this work and suggests directions for future work.

3.2 Exception Handling and Fault Tolerance

Following the terminology adopted by Lee and Anderson [38], 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 [9]. 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 when that service has been completed. 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 3.1). 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

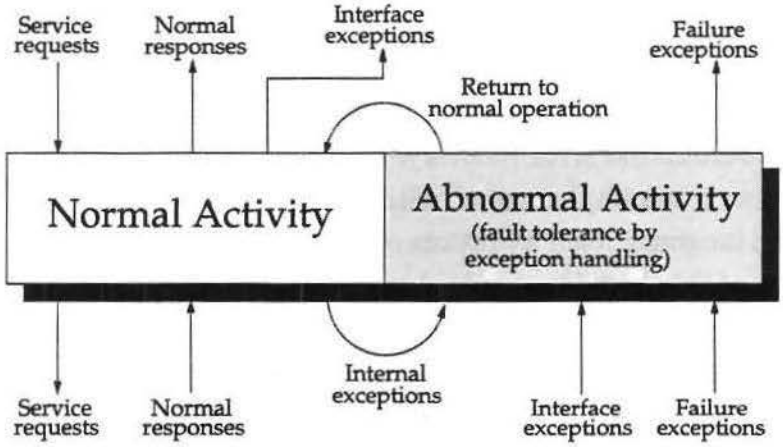


Figura 3.1: Idealized Fault-Tolerant Component.

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. At each level of the system, a component, called an *idealized fault-tolerant component* [38], will either deal with exceptional responses raised by components at a lower level or else propagate the exception to a higher level of the system.

Programmers usually refer to faults as exceptions because they are expected to occur rarely during the component's normal activity. *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 raised by a component, this mechanism is responsible for changing the normal control flow of the 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.

3.3 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 each of these issues 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 [44]. 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++ [35] and Java [30] 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() operation. In this case, the exception is a standard object that receives messages. The exception handling system implemented in Lore [15] applies this design solution.

Handler Attachment. Handlers can be attached to: (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 with 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 object 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 a more specific handler cannot 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: (i) *automatic*, or (ii) *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. There are essentially two possible solutions, which correspond to different styles of continuation of the control flow: (i) *termination*, and (ii) *resumption*. In the termination model, execution continues 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 execution has the capability to resume the internal activity of the component after the point at which the exception was raised.

Support for Coordinated Recovery. Very few object-oriented languages support concurrent exception handling, e.g. the activation of several handlers in different concurrent objects when an exception has been raised by one of them. For instance, the Arche language [32, 33] 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.

3.3.1 Exception Handling and the Object Model

Even though many object-oriented languages provide exception-handling facilities, only 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 provide 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 [46]. 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 signatures should also include the exceptional responses that an object may return. For example, Java [30] allows the declaration of the exceptions a method may signal in its signature with a clause throws. 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 writeability, it is necessary to allow 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 and can be applied to all methods of that class. It is also possible to attach handlers to objects themselves.

3.3.2 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 into three categories [38]: (i) *independent*, (ii) *competing*, or (iii) *cooperating*. Threads are said to be independent if the sets of objects accessed by them are disjoint; when those sets are not disjoint, then the threads are said to be competing. 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 competing threads, the provision of recovery is similar to the first case, but the set of objects accessed by the threads should be restored to an 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 in the presence of cooperative concurrency: the handling of an exception may involve multiple concurrent components when they are cooperating in the execution of a task. Erroneous information may have been spread directly or indirectly through inter-thread communication. When one of the concurrent threads raises an exception, error recovery should proceed in a coordinated way by triggering appropriate handlers for the same exception within all the threads [69].

Furthermore, due to the 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

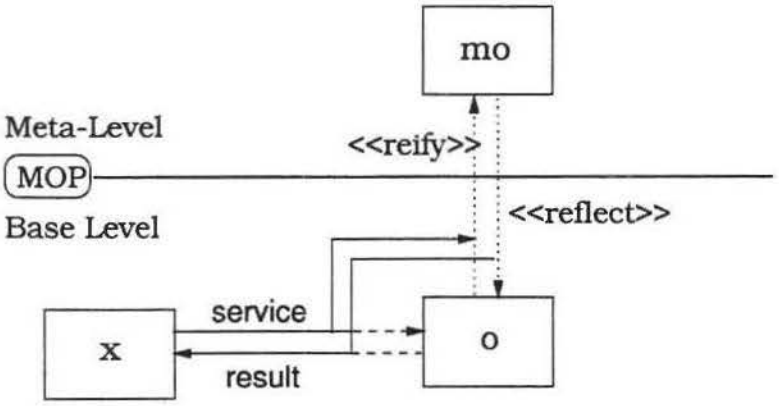


Figura 3.2: A Meta-Level Architecture

concurrently may be the symptom of a different and more serious fault [69]. In this way, an exception resolution procedure is needed to select a suitable handler for the exceptions raised concurrently; in this case, such a generic handler should also be called in all the threads. The work of Campbell and Randell [9] 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 a *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.

3.4 Reflection and Meta-Level Architectures

Computational reflection [43, 51] is defined as 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 at least two dimensions: (i) a *base level*, and (ii) 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 to the systems.

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

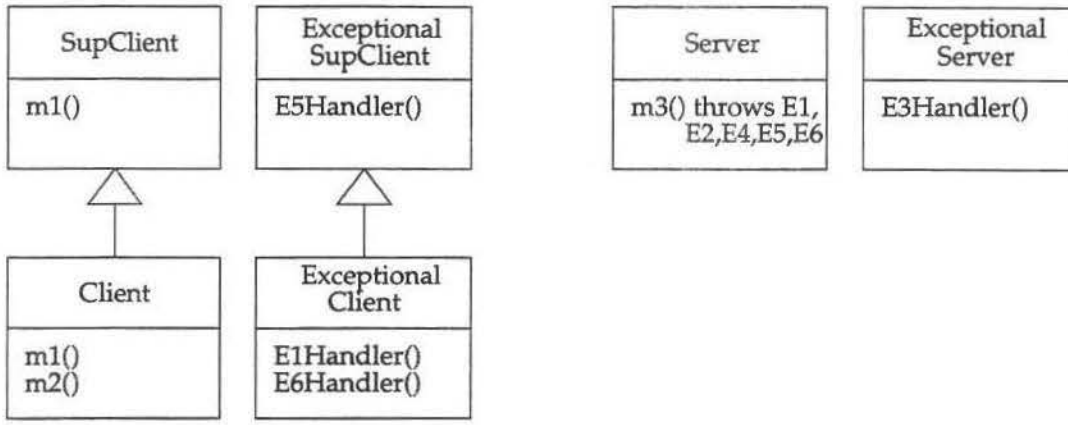


Figura 3.3: Normal and Exceptional Class Hierarchies

there exists a corresponding meta-object mo that represents the behavioral and structural aspects of o. As illustrated in Figure 3.2, 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 the 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 has been intercepted and redirected to the meta-level.

3.5 An OO Exception Handling Model

The exception handling 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 the design choices for each one of the major design issues described in Section 3.3.

As discussed in Section 3.2, 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 3.3). Therefore, exceptional classes implement the abnormal activity of the application and they are associated to the corresponding normal classes. In Figure 3.3, 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

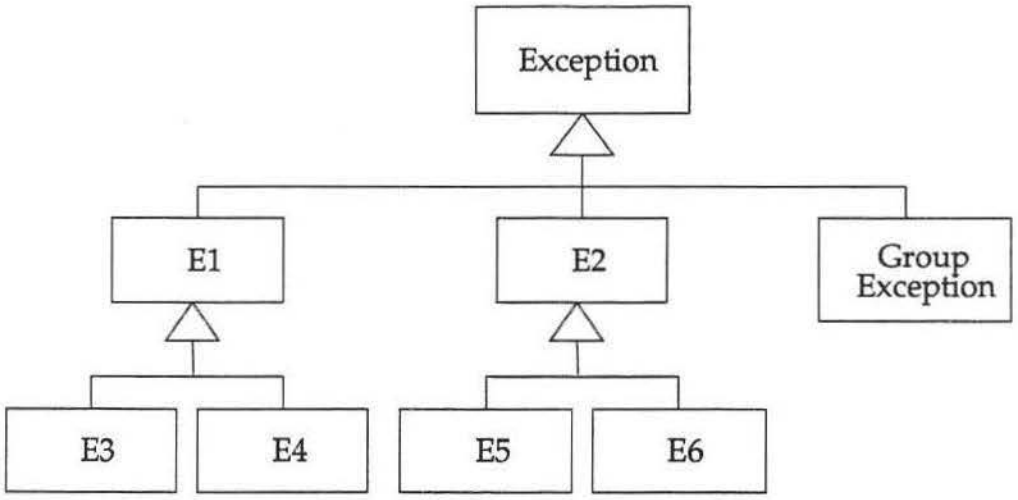


Figura 3.4: An Exception Class Hierarchy

hierarchy (SupClient and Client). Exceptional class hierarchies allow exceptional subclasses to inherit handlers from their superclasses and, consequently, they allow exceptional code reuse.

3.5.1 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 3.4 shows this exception class hierarchy which represents the exceptions that may be raised during the execution of the 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 3.5.5). Exceptional responses that may be signaled by a method must be described in its method's signature by means of a throws clause. Figure 3.3 shows that 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, which is a subtype of E. Permitting several exceptions to be named in the same handler avoids code replication when the exceptions can be handled in the same way.

3.5.2 Handler Attachment

We provide support for multi-level attachment of handlers. Handlers may be associated with: (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

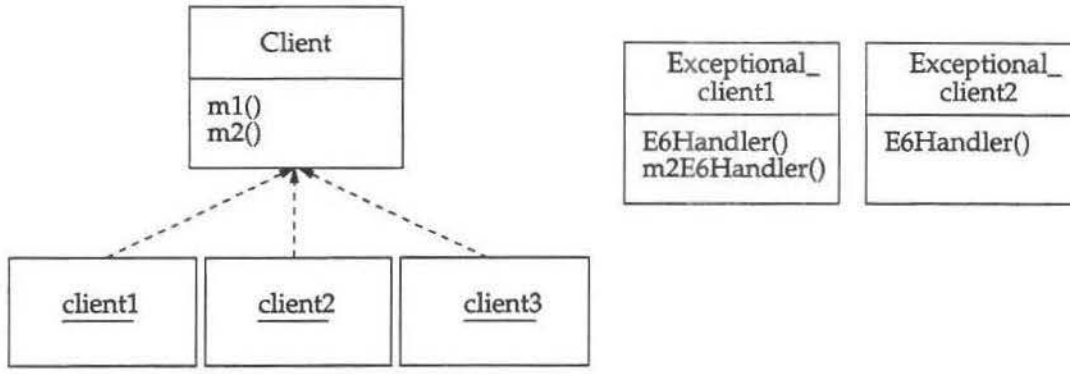


Figura 3.5: Objects and their Exceptional Classes

executed in the absence of a more specific handler in the application. Handlers may be also associated to a class. In this case, an exceptional class should be created. In Figure 3.3, the ExceptionalSupClient's methods are class handlers for the exceptions that should be treated within SupClient's methods. In the same way, 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 superclasses of the class ExceptionalClient. Therefore, the handler for the exception E5 (E5Handler()) is inherited from the ExceptionalSupClient.

In addition, object handlers may also be 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, 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 class Client (Figure 3.5). The Exceptional_client1's methods are object handlers for exceptions that should be treated within 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 [13] have shown that the use of object handlers can produce better structured programs, facilitating their understanding, maintenance and reuse. Finally, handlers may be associated to methods. For example, handler m2E6Handler() of exceptional class Exceptional_client1 is activated when the exception E6 should be treated in operation m2().

The search of handlers for raised exceptions is defined as follows: (i) if there 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 of these is found, the exception is then signaled to the caller object and steps (i) e (ii) are repeated; (iv) still, 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 exceptions E1, E2, E4, E5 or E6. Suppose m3() signals E1 to m1(); then class handler E1Handler() of ExceptionalClient is invoked. In case m3() signals E4, class handler E1Handler() is also invoked since E4 is subtype of E1 (Figure 3.4). If m3() signals E5, class handler E5Handler(), inherited from ExceptionalSupClient, is invoked. In case m3() signals E6 to m1(), object handler E6Handler() of Exceptional_client1 (Figure 3.5) is invoked in spite of the presence of the class handler. Suppose m3() is invoked by m2() of object client1, if m3() signals E6; then method handler m2E6Handler() of Exceptional_client1 is invoked.

3.5.3 Exception Propagation

Our exception handling model defines explicit propagation of exceptions. The benefits of this approach are discussed in [12, 73]. 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 resignaled 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 [13]. The CLU designers [40] argue persuasively that this limitation supports the goals of good program structuring with only a minor loss in its writeability.

3.5.4 Continuation of the 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 simpler and more suitable for construction of dependable systems [10]. Mechanisms that support resumption are very powerful and flexible, but they turn out to be difficult to use by application programmers. In fact, they can promote the unsafe programming practice of removing the symptom of an error without removing the cause.

3.5.5 Support for Coordinated Recovery

Since the cooperating activities are application-dependent, 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 their concurrent object-oriented systems, and supporting coordinated recovery. In this sense, coordinated recovery only needs to be activated within the participant threads of a group. This obviously restricts system design but makes it possible to regard each *group* as a recovery region and attach fault tolerance activities to each group *participant*. We enable the definition of *subgroups* which contribute to control the system complexity and allow better organization of both normal and abnormal activities of the enclosing group.

Figure 3.6 shows threads, represented as lines, and activities of the groups, delimited by rectangles. Group B, composed by threads T2, T3 and T4, is a subgroup of group A which has the same composition as B, added of T1. After the occurrence of an exception in one of these threads (T3), other participants of the same group (T2 and T4) should be informed 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, the group activity must be undone (backward error recovery), and such an exception must be signaled to the enclosing group (group A). If 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 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. Each group participant has a set of attached exception handlers that are designed to recover the group cooperatively from eventual errors. An exception tree (Section 3.3.2) is associated to each group in order to resolve the exceptions raised concurrently.

Implementation 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 which deal with the creation and termination of each participant. Secondly, the programmer should define the participants that compose the group by extending the class Participant. Figure 3.8 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 the group activity is started. In order 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 to work cooperatively. A new class that extends

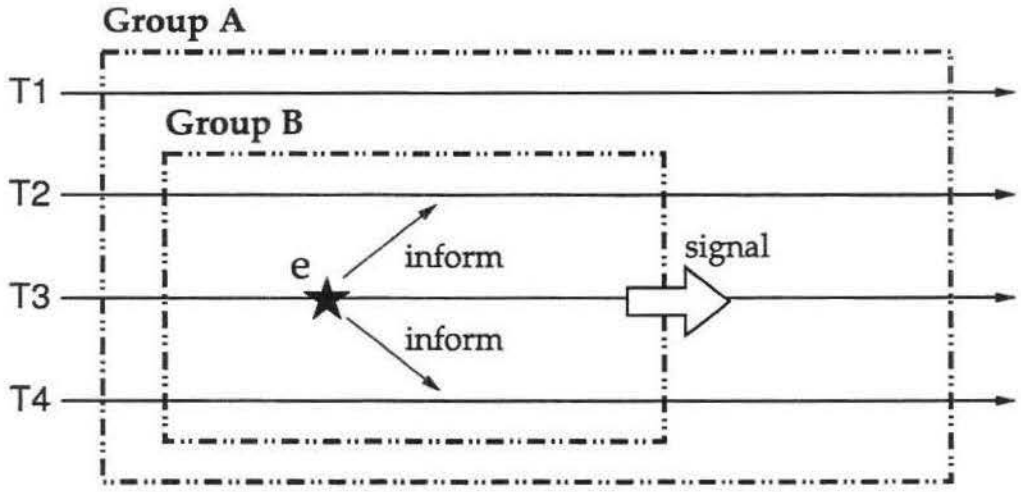


Figura 3.6: Exception Propagation

the class Group must also be instantiated (group1) before the group activity commences. In building an instance of this class, the following parameters should be passed: (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 to the enclosing group. The participants still may register themselves dynamically in a group through the method RegisterParticipant (Participant) of the class Group. This class still provides the method StartParticipant (Participant) which allows a participant to enter dynamically in a group activity.

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 [20] (Figure 3.8) to define simple and structured exceptions. This pattern allows application designers to treat simple exceptions and its compositions (structured exceptions) uniformly. Simple exceptions are defined by extending class GroupException (E1 and E2). Structured exceptions are instances of 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.

3.6 Twin-Engine Aircraft Control System

This Section highlights the benefits of the proposed exception mechanism for the design of reusable and dependable object-oriented software. We present a twin-engine aircraft

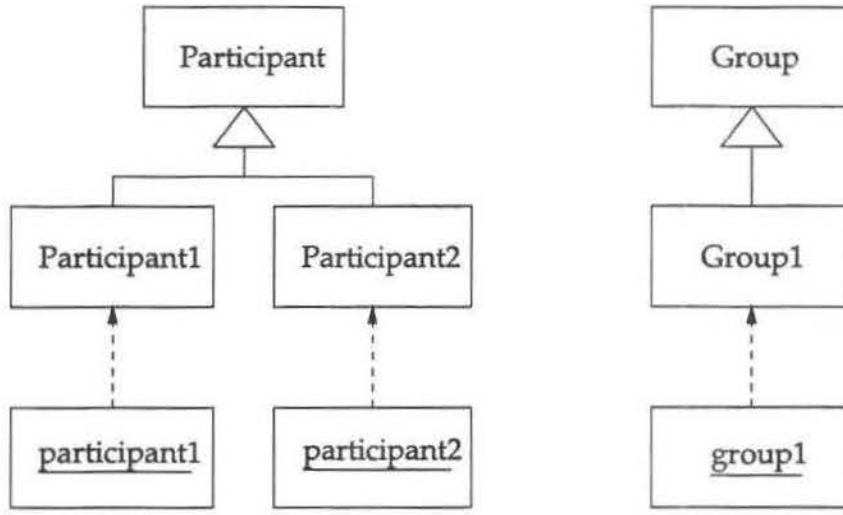


Figura 3.7: The Definition of a Group

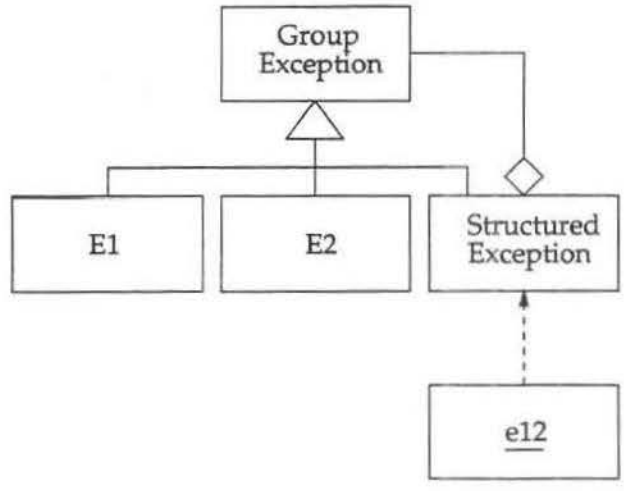


Figura 3.8: The Definition of the Exceptions for a Group

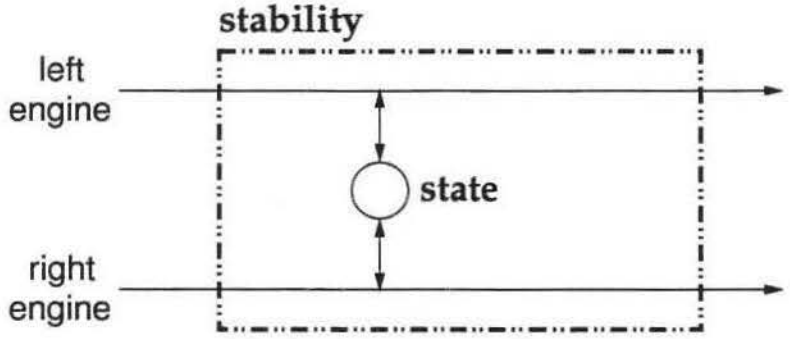


Figura 3.9: The Cooperating Activity of the Group Stability

control system that is based on the example described in [9]. 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 to maintain the aircraft stability.

Figure 3.9 shows the participants left_engine and right_engine of 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 3.10. 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 the exception mechanism that searches the handlers for the structured exception emergency_exception attached to the 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 3.11 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. In order to start the group activity, two instances of the class Engine and one of the class Stability must be created. Participants execute object methods for performing the group activity. In this example, the participants left_engine and right_engine execute the methods of objects left_control and right_control when performing the cooperative group activity.

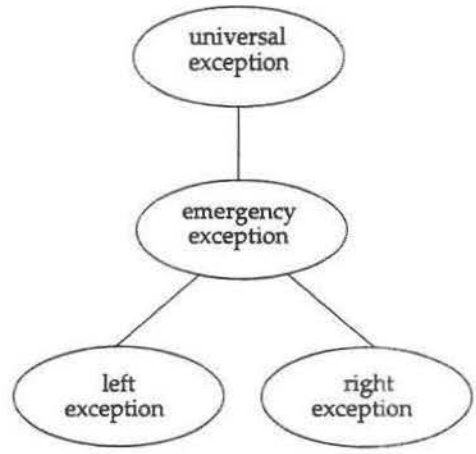


Figura 3.10: The Exception Tree of the Group Stability

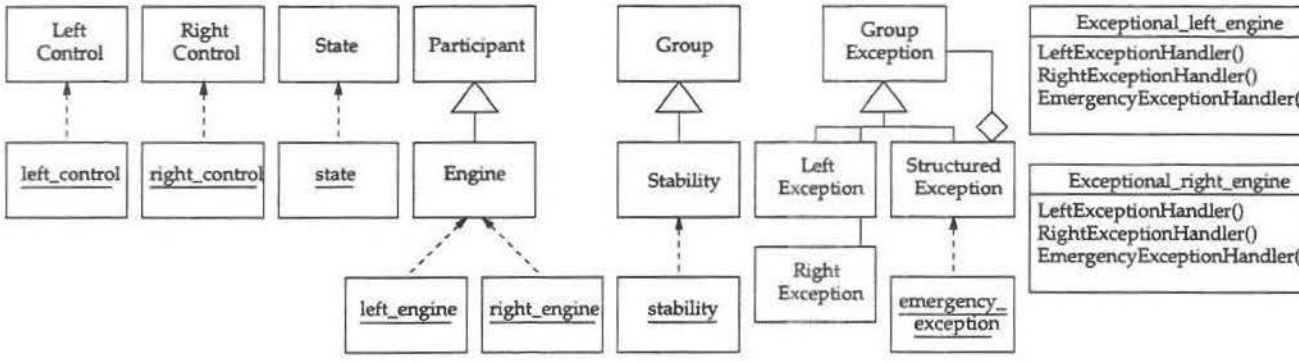


Figura 3.11: Object Model for the Twin-Engine Aircraft Control System

The purpose of object state, instance of class State, is communication between the cooperating participants. Simple exceptions LeftException and RightException and structured exception emergency_exception are also defined. Exceptional classes Exceptional_left_engine and Exceptional_right_engine contain the methods which 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 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 (left_exception,
                                                right_exception);
```

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

```
(1) Object[] Participants = {left_engine, right_engine};
(2) Object[] InternalExceptions = {left_exception, right_exception};
(3) Object[] ExternalExceptions = {emergency_exception};
(4) Stability stability = new Stability (Participants, InternalExceptions,
                                                    ExternalExceptions);
```

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 to the enclosing group. Line 4 creates the object that represents the group Stability.

3.7 Implementation

3.7.1 The Meta-Level Architecture

In this section, we present a meta-level software architecture for implementing our 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 its corresponding meta-object searches for an adequate handler in a way that is transparent to the application at the base level. Applications are composed of normal classes that implement the normal functionality and exceptional classes with handlers for the corresponding normal classes.

Figure 3.12 illustrates the meta-level architecture for implementing the exception mechanism. The base level is composed of: (i) the exception class hierarchy (Figure 3.4); (ii) normal class hierarchies (Figure 3.3); (iii) exceptional classes with handlers that are associated to normal classes (Figure 3.3) and (iv) exceptional classes with handlers that are associated to objects (Figure 3.5).

The meta-level is composed of: (i) composers, and (ii) meta-searchers. The composers are special meta-objects associated to the application's objects or classes. They delegate information from the base-level to meta-objects responsible for several management actions, such as exception handling, persistency and atomicity. The meta-searchers are meta-objects responsible for managing exception handling. Furthermore, they receive information 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

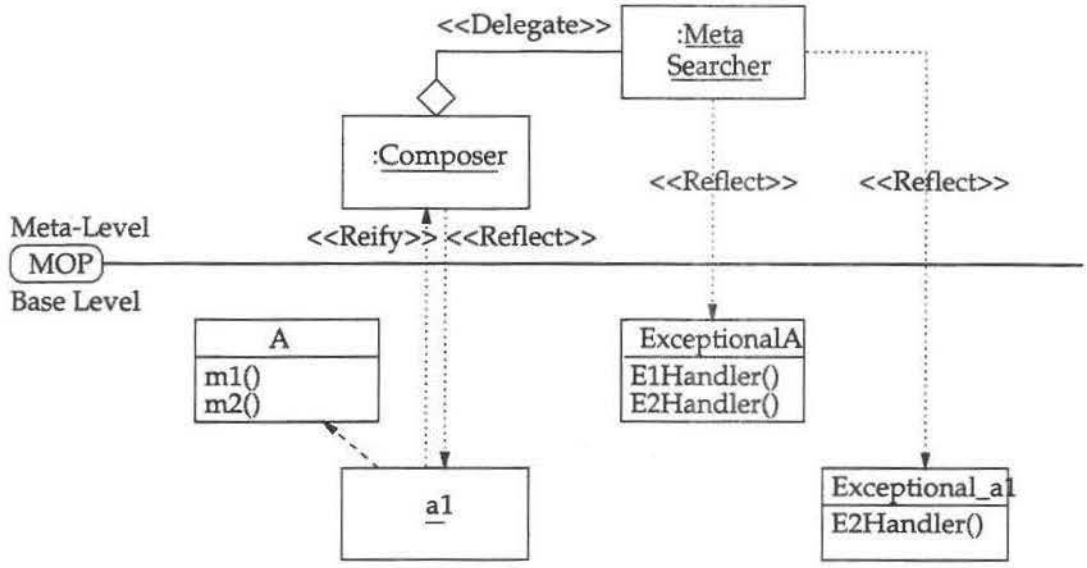


Figura 3.12: The Proposed Meta-Level Architecture.

to the raised exception; (ii) invocation of the handler; (iii) return to the normal operation of the application.

3.7.2 The Meta-Level Architecture and Concurrency

Figure 3.13 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) [49]. Each instance of Participant (Section 3.5.5) is associated to a composer and a metaseacher; each instance of Group (Section 3.5.5) 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 of 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 to the enclosing group.

The meta-group sends the simple and structured exceptions to EPS that must compose the group's 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

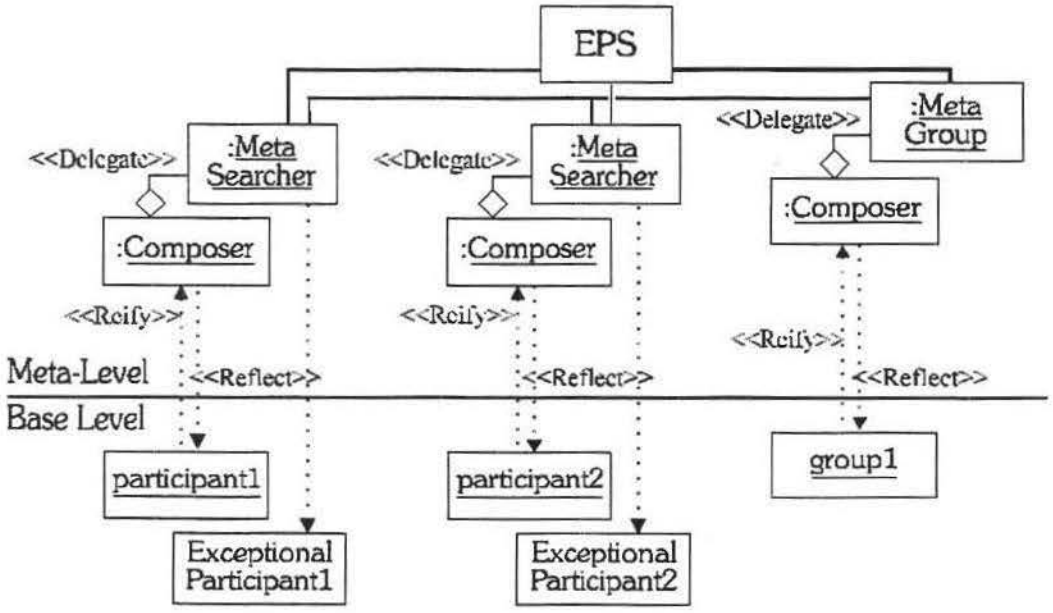


Figura 3.13: The Meta-Level Architecture for Concurrency.

are activated in a way that is transparent to the application.

3.7.3 Implementation Issues

Our mechanism does not require any special language support, and it was implemented within the Java programming language. Moreover, EPS has allowed the construction of the composition scheme of exception trees based on the aggregated tree concept [49], 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* [51]. Guaraná is a flexible meta-object protocol for Java that allows creating meta-level objects. Guaraná provides an efficient broadcast service for communication between meta-objects. Moreover, it provides support for composition of meta-objects responsible for different management functions by means of composers. These Guaraná capacities and the way our exception mechanism was de-

signed allow the meta-objects of our exception handling system to be easily integrated with meta-objects responsible for other administrative (non- functional) services, such as persistency and atomic actions.

3.8 Related Work

The work of Hof *et al.* [31] describes an exception mechanism based on meta-programming and computational reflection. Their implementation was carried out in a specific system but it could be implemented to most other systems that support meta-programming. However, such a mechanism does not support coordinated recovery in concurrent threads and its design is not object-oriented.

The Arche language [32, 33] 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.

3.9 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 software components. This separation allows the production of 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 the reuse of exceptional code. Moreover, the design of our mechanism is integrated with object paradigm and provides support for 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 of activities related to management of exception handling from the exceptional and normal activities of the application.

The Coordinated Atomic Action concept (CAAction) [69] was introduced as a unified approach for structuring complex concurrent activities and for 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 the off-the-shelf approach to object-oriented software development, achieved by selecting and configuring reusable components, has resulted in a significant decrease of development costs. In this work, we have designed a mechanism that supports the construction of reusable and dependable software components. Still, an open issue is how to allow that exception-handling code to be added to reusable components (for instance, COTS) without any interference in the original code of these components. This additional exception- handling code should handle the new exceptions that can arise when these components are reused in different applications.

3.10 Resumo do Capítulo 3

Este capítulo apresentou um artigo que aborda o projeto e implementação de um mecanismo de exceções para construção de software orientado a objetos confiável. O modelo de tratamento de exceções permite uma separação explícita entre as atividades normais e excepcionais de aplicação, fundamental para manter a complexidade de sistemas confiáveis sob controle. Essa separação contribui efetivamente para a produção de componentes de software que são fáceis de entender, reutilizar e manter. O modelo de tratamento de exceções é orientado a objetos e provê suporte para tratamento de exceções concorrentes.

O mecanismo de exceções foi implementado para a linguagem Java sem modificações para a mesma através da utilização da arquitetura de software reflexiva do Guaraná [51]. A utilização de reflexão computacional permitiu uma divisão clara entre as funcionalidades da aplicação e os serviços do mecanismo de exceções proposto, resultando na construção de um mecanismo de exceções simples e fácil de usar.

O próximo capítulo apresenta uma arquitetura de software reflexiva para o projeto de mecanismos de exceções e o conjunto de padrões de projeto que documentam os componentes da arquitetura proposta.

Capítulo 4

Uma Arquitetura de Software Baseada em Padrões para Mecanismos de Exceções

A arquitetura de software de um sistema compreende os componentes computacionais e as interações entre estes componentes, definindo também a relação entre os requisitos e os elementos de software [65]. Padrões de projeto constituem boas soluções de projeto para problemas recorrentes dentro de um contexto particular [7, 20]. Padrões de projeto identificam soluções existentes e bem provadas, e a documentação destes padrões facilita o entendimento destas soluções.

Este capítulo contém o artigo *"An Exception Handling Software Architecture for Developing Robust Software"* [22], que foi submetido para *"5th IEEE International Symposium on High Assurance Systems Engineering"*, a ser realizado de 15 a 17 de novembro de 2000, em Albuquerque, New Mexico, México. Uma versão resumida [24] deste artigo foi aceita para o *"2nd Workshop on Exception Handling in Object-Oriented Systems – ECOOP'2000"* a ser realizado em 12 de junho de 2000, em Cannes, França. Este artigo define uma arquitetura de software reflexiva para mecanismos de tratamento de exceções que serão utilizados na construção de sistemas orientados a objetos confiáveis. Além disso, este artigo propõe padrões de projeto que são aplicados para documentar a estrutura e o comportamento dos componentes arquiteturais de um mecanismo de tratamento de exceções.

An Exception Handling Software Architecture for Developing Robust Software

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

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

4.1 Introduction

Modern object-oriented software systems are getting more complex and have to cope with an increasing number of error conditions to meet the system's dependability requirements. Dependable object-oriented software detects errors caused by residual faults and employs fault tolerance measures to restore normal computation [38]. Exception and exception handling provide a suitable scheme to detect and handle errors, and also incorporate fault tolerance activities into software systems. The detection of an error will result in an exception being raised, with an appropriate handler corresponding to the raised exception being automatically invoked to implement the fault tolerance measures [38]. The presence of *exception handling facilities* can reduce software development efforts since they allow software designers to: (i) represent errors as exceptions, (ii) define handlers to deal with them, and (iii) use an adequate strategy for exception handling when the occurrence of an exception is detected.

Moreover, object-oriented systems may be consisted of various execution threads (or processes) executing methods concurrently on objects. Exceptions are more difficult to handle and exception handling facilities to provide in concurrent object-oriented systems than in sequential ones specially because of cooperative concurrency [9]. That is, several concurrent threads usually cooperate to perform some system's activity, giving rise to very complex concurrent interactions. In this context, erroneous information may be spread directly or indirectly through inter-thread communication during a cooperative activity. A general approach for structuring cooperative activities and employing exception handling in concurrent systems extends the well-known atomic action notion [9]. An atomic action is formed by a group of participants which are executed by cooperating threads. The group cooperate within the scope of an action and complex interactions are coordinated by the action, including the management activities related to concurrent exception handling.

Participants may join an action asynchronously but they have to leave it synchronously to guarantee that no information is smuggled to or from the action. When an exception is raised in any of the participants inside an action, all action participants should participate in the error handling [9]. In general, different exception handlers for a same exception have to be called in the participants. These handlers are executed concurrently in order to handle the exception in a coordinated way. An additional difficulty is that several exceptions can be raised concurrently by participants during a cooperative activity. In this situation, a process of exception resolution is required to agree on the exception that should be notified to all participants.

Exception handling facilities for sequential programs are usually incorporated in various modern object-oriented programming languages, such as C++ [35], Java [30] and Eiffel [45]. However, very few languages give direct support to concurrent exception handling (for instance, Arche [33]); but, in general, the solutions presented cope with concurrent exception handling in a rather limited form. Recently some 'ad hoc' solutions have been proposed to the provision of concurrent exception handling which extends programming languages, such as Ada and Java [58, 72, 74]. However, we believe that these recent proposals present complex solutions which are also very language-dependent and error prone. Besides, these solutions can be very intrusive from the viewpoint of the application since its normal code is usually amalgamated with explicit references and invocations of procedures responsible for exception resolution and final synchronization of the action participants. In addition, the task of software developers is also complicated in the sense that they have to implement exception resolution functions for each cooperative activity of the system. Consequently, these solutions present exception handling techniques which are difficult for software developers to use, and may produce software products which are non-reliable and difficult to understand, maintain and reuse.

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

(iv) the *Concurrent Exception Handling Action* component. The structural and behavioral aspects of the components are described by means of a set of design patterns. The patterns follow the overall structure of the *Reflection* architectural pattern which allows a clear and transparent separation of concerns between the application's functionality and the exception handling facilities, easing the task of building robust software.

The remainder of this text is organized as follows. Section 4.2 introduces a number of concepts and difficulties related to exception handling, and also presents the general abstraction for exception handling facilities. Section 4.3 presents object-oriented techniques for design reuse and software structuring used for the development of the proposed solution. Section 4.4 shows the proposed software architecture for exceptional condition handling. Section 4.5 presents the set of design patterns for exception handling. Section 4.6 discusses some implementation issues. Section 4.7 gives a brief comparison with related work. Finally, Section 4.8 summarizes the conclusions of this work and suggests directions for future work.

4.2 Exception Handling

4.2.1 Exception Handling in Sequential Systems

Developers of dependable systems usually refer to errors as exceptions because they are expected to occur rarely during a system's normal activity. These exceptions should be specified internally into the system and an instance of an exception raised at run-time is termed an *exception occurrence*. Some extra-information about an exception occurrence, such as its name, description, location, and severity [37], is usually required by an application, and it is useful for handling an exception occurrence. Extra-information is passed either explicitly by the application component that has raised the exception, or implicitly by an exception handling service.

Dependable applications need to incorporate exception handling activities in order to behave suitably in a great number of exceptional situations. Exception handling activities are structured by a set of *exception handlers* (or simply *handlers*). A handler is the part of an application code that provides the measures for recovering the system from a detected exception. A handler may be valid for one or more exceptions. Handlers are attached to a particular region of normal code which is termed a *protected region*. Each protected region may have a set of attached handlers, and one of them is invoked when a corresponding exception is raised. Handlers can be attached to *blocks of statements*, *methods*, *objects*, *classes*, or *exception classes*. Handlers attached to exception classes, called *default handlers*, are the most general handlers, and must be valid in any part of the program, independently of any execution context and object state. For the purpose

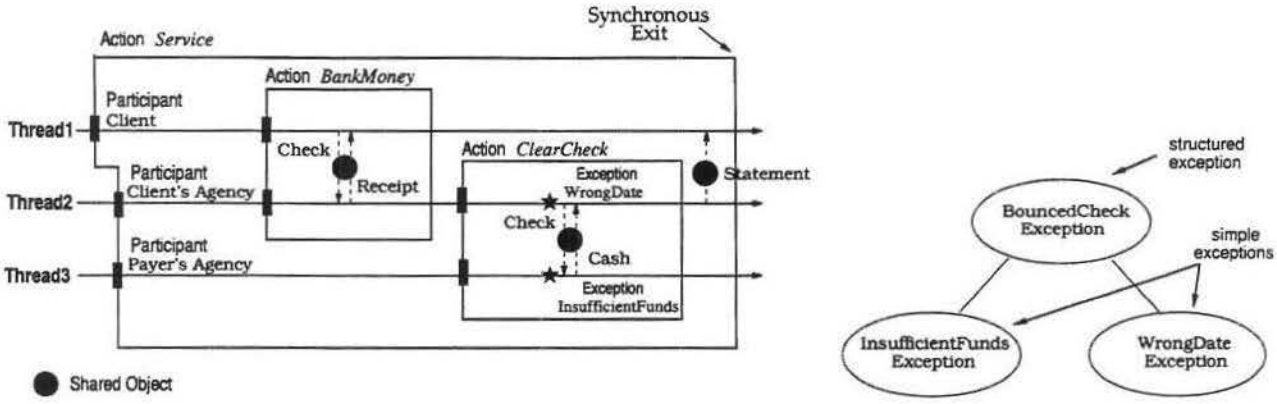
of improving the writeability and structuring of the software systems, it is desirable to allow some flexibility concerning the attachment of handlers. It is should be possible the *multi-level attachment of handlers*, i.e., the attachment of handlers to several levels of protected regions such as classes, objects, methods and so on.

An exception handling strategy should be followed after an exception occurrence is detected. In general, the *normal control flow* of the computation is deviated to the *exceptional control flow*. The deviation of the control flow is followed by the search for a suitable handler to deal with the exception occurrence. The handler search is performed according to a search algorithm. When a handler is found, it is invoked and the computation is returned to its normal control flow. The returning point where the normal flow continues also depends on the chosen model for the continuation, namely, the *termination* model, or the *resumption* model. In the termination model, execution continues 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 execution has the capability to resume the internal activity of the component after the point at which the exception was raised. The semantic of the termination model is simpler and more suitable for construction of dependable software [10].

4.2.2 Exception Handling in Concurrent Systems

In this work, cooperative activities of a dependable concurrent object-oriented system are structured as a set of atomic actions. We refer to these activities to as *concurrent cooperative actions* (or simply *actions*). An action provides a mechanism for performing concurrently a group of methods on a collection of objects. The interface of an action includes its participants and methods (and their respective objects) that are manipulated by the participants. In order to perform an action, a group of threads should execute each participant in the action concurrently (one thread per participant). Threads participating in an action cooperate within the scope of the action by executing methods on objects, and exchange information only among ones that are participants of that action. Threads cooperate and communicate each other by means of *shared objects*. The entries of participants in the action may be asynchronous but they have to leave the action synchronously to guarantee that no information is smuggled to or from the action.

We introduce a banking service example based in [9] that illustrates the concepts of concurrent exception handling. This example is also used throughout Section 4.5 to illustrate how our proposed approach can be employed. Figure 4.1(a) shows the structuring of concurrent cooperative actions in the banking service example. Threads participating in the action are represented by solid lines, inter-thread communication by dotted lines, and actions by rectangles. Action participants are activated by threads which cooperate



(a) Cooperating Threads (b) Exception Tree

Figura 4.1: Banking Service Example.

within the action's scope for performing the banking service. The participants of the action Service are Client, Client's Agency and Payer's Agency. Consider a Client that presents a check (i.e., an object of the type Check) to his/her bank and receives a Receipt that certifies the operation. To clear the check, the Client's Agency sends the Check to Payer's Agency which has the payer's account. Once Client's Agency receives the Cash for the check, it sends to Client a new Statement of his/her account. Actions can be nested and exceptions may be propagated over nesting levels. In any moment, some action participants can start nested actions. Figure 4.1(a) shows two nested actions for the action Service. The participants Client and Client's Agency perform the nested action BankMoney, and the participants Client's Agency and Payer's Agency perform the nested action ClearCheck.

Exception occurrences can be raised by participants during an action. Some of them can be handled internally by a local handler attached to the participant that raised that exception. We refer to these exceptions as *local exceptions.* Traditional exception handling strategies address this kind of exception. If an exception occurrence is not handled internally by a participant, then it should be handled cooperatively by all action participants. This kind of exception is called a *cooperating exception*, and, in this case, a new concurrent exception handling strategy is required. When a cooperating exception is raised in any of the participants inside an action, all action participants have to participate in its handling. So, a set of cooperating exceptions is associated with each action. Each participant has a set of handlers for (all or part of) these exceptions. Participants are synchronized and probably different handlers for the same exception have to be invoked in all participants [9]. These handlers are executed concurrently, and cooperate to

handle the cooperating exception in a coordinated way. Moreover, various cooperating exceptions may be raised concurrently while participants are cooperating in the action. So, a mechanism of *exception resolution* is necessary in order to agree on the cooperating exception to be notified to all participants of the action. The paper [9] describes a model for exception resolution called *exception tree* which includes an exception hierarchy. If several cooperating exceptions are raised concurrently, the resolved exception is the root of the smallest subtree containing all raised exceptions. Cooperating exceptions can be of two different kinds in the exception tree: (i) *simple exceptions*, or (ii) *structured exceptions*. Simple exceptions are leafs of the tree and correspond to cooperating exceptions being raised alone concurrently. Structured exceptions are non-leaf nodes and correspond to two or more simple exceptions being raised concurrently. An exception tree should be specified for each action of the application. In Figure 4.1(a), during the action ClearCheck, two cooperating exceptions are raised concurrently, namely WrongDateException and InsufficientFundsException. Figure 4.1(b) presents the exception tree specified for the action ClearCheck. The structured exception BouncedCheckException represents the concurrent raising of the simple exceptions WrongDateException and InsufficientFundsException.

Participants of an action can leave it on three occasions. First of all, they can leave the action if no exceptions were raised. Secondly, if cooperating exceptions have been raised, but handlers have successfully handled them. Thirdly, they can leave the action signaling a *failure* exception to the containing action if a cooperating exception has been raised and no proper handlers were found or the handling of that exception is not possible. There are at least two distinct approaches for concurrent exception handling: (i) the *blocking* approach, and (ii) the *pre-emptive* approach. In blocking schemes, each participant terminates by reaching the end of an action or fails by raising a cooperating exception. Participants are informed of an exception occurrence only when they are completed (or detect a cooperating exception); that is, when they are ready to accept information about the state of other participants. In contrast, pre-emptive schemes do not wait but require some language feature to interrupt all participants when cooperating exceptions are raised [59]. In blocking systems, exception handling and resolution are easier to provide than in pre-emptive ones because each participant is ready for handling when handlers are invoked. Moreover, there is no need to perform the abortion of nested actions because they have either been completed successfully or have had exceptions dealt by nested action's handlers.

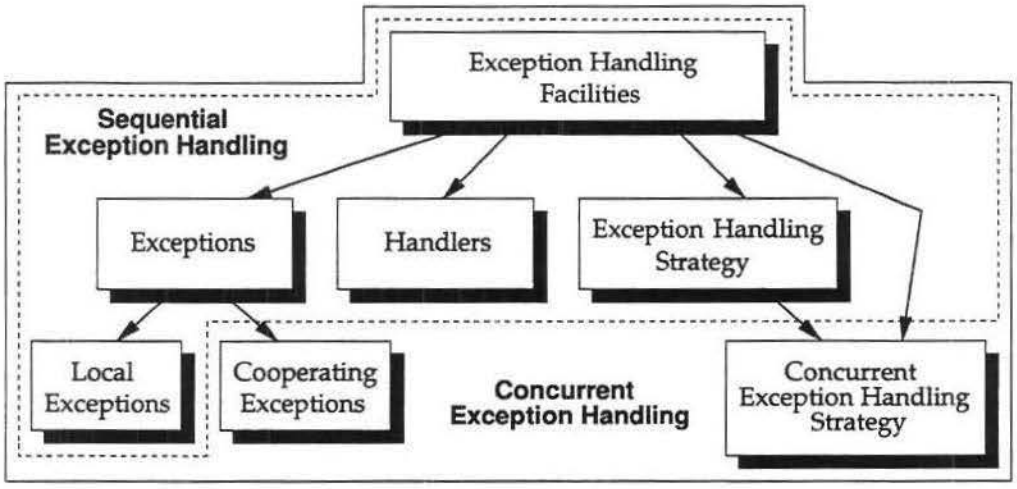


Figura 4.2: Integration of Exception Handling

4.2.3 Integration of Sequential and Concurrent Exception Handling

Figure 4.2 illustrates the integration of sequential and concurrent exception handling. Sequential exception handling facilities include: (i) exceptions - the definition and raising of local exceptions, and management of extra-information about exception occurrences, (ii) handlers - the definition and invocation of handlers, and (iii) exception handling strategy - the specification of an algorithm for handler search, and a model for continuation of the control flow. As discussed earlier, concurrent exception handling requires some extra support not required by sequential systems. So, an integrated approach to exception handling should support both local and cooperating exceptions, and also a concurrent exception handling strategy. Ideally the concurrent exception handling strategy should be consistent with the exception handling strategy (of the sequential exception handling). In this work, the strategy for concurrent exception handling extends the atomic action paradigm described previously.

4.3 Design Reuse and Software Structuring Techniques

4.3.1 Software Architecture and Patterns

A system's software architecture abstractly describes the system's gross organization in terms of *components* and their interrelationships [65]. Components are physical and replaceable parts of a architecture, and to each component are attached responsibilities. The

components must interact with each other in the described fashion, and each component must fulfill its responsibilities to the other components as dictated by the architecture. Each component conforms to and provides the realization of a set of *interfaces* [3]. The interfaces make available services which are implemented by the component.

Software patterns are an important vehicle for constructing high-quality architectures [2]. Patterns are useful mental building-blocks for dealing with limited and specific design aspects when developing a software architecture. Patterns are discovered rather than invented, and they exist in various ranges of scale. *Architectural patterns*, for instance, define the basic structure of an architecture and systems which implement that architecture [7]. *Design patterns* are however more problem-oriented than architectural patterns, and are applied in later design stages. Usually, the selection of a design pattern is influenced by the architectural pattern that were previously chosen. A design pattern expresses a very specific recurring design problem and presents a solution to it, all from the viewpoint of the context in which the problem arises [7]. Moreover, a design pattern must balance, or trade off, a set of opposing forces. Design patterns refine the general components of an architecture, providing the detailed design solutions.

In this work, each component of the proposed architecture implements a design pattern which describes the design of the corresponding component. The proposed architecture's components and their corresponding design patterns follow the overall structure of the *Reflection* architectural pattern [7]. This pattern captures the benefits from computational reflection and meta-level architectures which are described in the next section.

4.3.2 Meta-Level Architectures and Computational Reflection

Computational reflection is a technique that allows a system to maintain information about itself (*meta-information*) and use this information to adapt its behavior [43]. This information is obtained by means of a process called *reification*. Reification is the representation of abstract language concepts such as classes and methods in form of objects. In the object model, reflection establishes a *meta-level architecture* which achieves a *separation of concerns* between applications and management mechanisms by extending transparently the semantics of the underlying system. Meta-level architectures are composed of at least two dimensions: (i) a *base level* (or application level), and (ii) a *meta-level* (management level). The base-level encompasses the objects responsible for implementing the functionality of the application. The meta-level encompasses the objects that deal with the processing of meta-information and management activities of an application. The meta-level objects (*meta-objects*) maintain structural and behavioral information of application objects. A meta-object protocol (*MOP*) establishes an interface of communication between base-level and meta-level objects. MOP provides a high-level interface

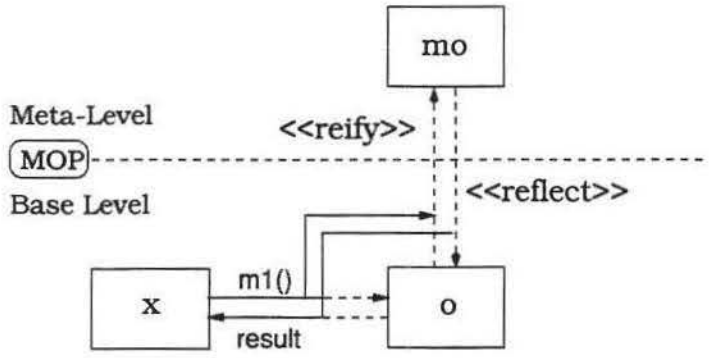


Figura 4.3: A Meta-Level Software Architecture.

to the programming language implementation in order to reveal the program information normally hidden by the compiler and/or run-time environment [43]. As a consequence, programmers can develop language extensions without any change to the programming language.

Computational reflection can be used to intercept, verify and modify transparently 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 4.3, if an object x invokes a method m1 on an object o, MOP intercepts this invocation, *reifies* the base-level computation and the meta-object mo takes over execution; later mo returns (*reflects*) the result to x. From the point of view of the object x, computational reflection is transparent: x sends a message requesting a method to o, and receives the result with no knowledge that the message was intercepted and alternatively altered by the meta-object.

4.4 The Software Architecture for Exception Handling

4.4.1 The Basic Architecture

This section presents a generic software architecture that integrates sequential and concurrent exception handling (Figure 4.4). Applications reuse our architecture to handle their exceptional situations by using the exception handling facilities provided by the architecture's components. The architecture is composed of four components: (i) the *Exception* component, (ii) the *Handler* component, (iii) the *Exception Handling Strategy* component, and (iv) the *Concurrent Exception Handling Action* component. Table 4.4.1 summarizes the components and their responsibilities. The responsibilities are classified into two kinds: (i) *application-dependent* responsibilities (ADR), and (ii) *application-independent*

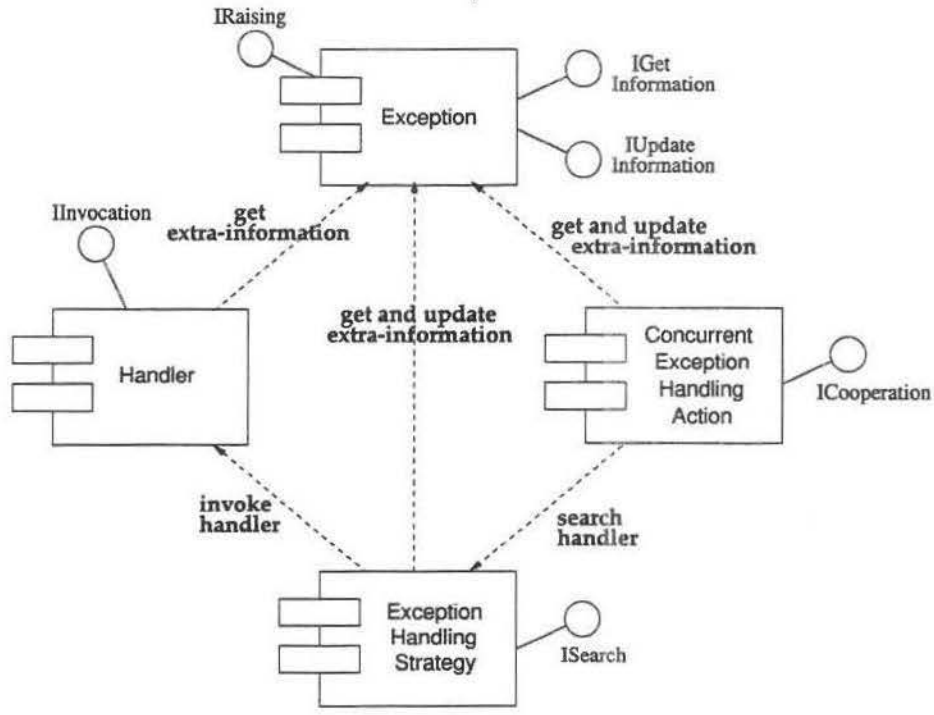


Figura 4.4: The Software Architecture for Exception Handling

responsibilities (AIR).

Application-dependent responsibilities are directly related to the application's functionality and include, for instance, facilities for specification of exceptions and handlers, raising of application exceptions, and specification of concurrent cooperative actions. The achievement of these responsibilities is application-dependent. As a consequence, the architecture's components provide the application developers with appropriate support in order to fulfill their application-dependent responsibilities. Developers of applications either invoke services provided by the architecture's component interfaces (Section 4.4.2), or else refine the design of architecture's components according to their needs (Section 4.4.3). For instance, application's components invoke the service provided by the *Exception* component in order to raise an application exception. Application designers tailor the *Exception*, *Handler* and *Concurrent Exception Handling Action* components to specify respectively exceptions, handlers and concurrent cooperative actions of their applications (Section 4.4.3). Exceptions, handlers and concurrent cooperative actions are part of the application's functionality.

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

Table 4.4.1: Components and their Responsibilities.

Application-independent responsibilities include, for instance, facilities for extra-information management, handler invocation, deviation of the control flow, handler search, participant synchronization and exception resolution. These responsibilities are related to management activities of exception handling. Components of our proposed architecture perform their management activities in a way that is transparent to the application (Section 4.4.3). As a result, the application developers concentrate their attention to the application's functionality and reuse the management activities for exception handling defined by the architecture. The architecture's components interact with each other as prescribed by the architecture in order to fulfill their application-independent responsibilities.

Figure 4.4 pictures the components and their interrelationships. The *Exception* component works as an extra-information holder component. It keeps extra-information about application exceptions which are used by the other components to achieve their responsibilities. Then the other components interact with the *Exception* component in order to get and update extra-information about exception occurrences. The *Exception Handling Strategy* component implements the services related to the general strategy for exception handling. Its responsibilities are the deviation of the control flow and the search for handlers. Therefore, this component plays a central role in the architecture and interacts with all other components. It asks the *Exception* component to provide extra-information about an exception occurrence while searching for its corresponding handler. After handler is found, it asks the *Handler* component to invoke the exception handler. The *Exception Handling Strategy* component also interacts with the *Concurrent Exception Handling Action* component. The later uses the services provided by the former in order to carry out the strategy for concurrent exception handling. For example, if one or more cooperating exceptions have been raised during a action, and the exception resolution has been accomplished by the *Concurrent Exception Handling Action* component, it asks the *Exception Handling Strategy* component to search the different handlers for the resolved exception.

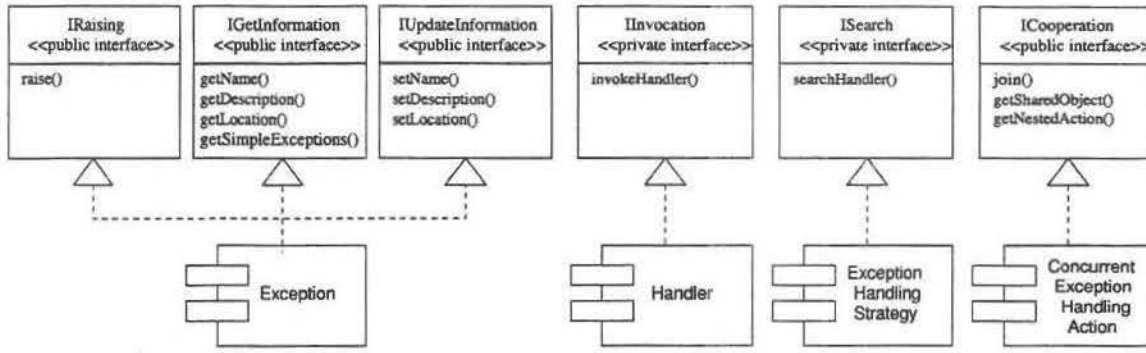


Figura 4.5: The Detailed Interfaces.

4.4.2 Interfaces of the Components

The interfaces of the components provide the exception handling services provided by the architecture's components. The interfaces are accessed either by the architecture's components themselves, or by the application while using the exception handling services. Figure 4.4 illustrates the architecture's components and their interfaces. The interfaces are classified in two sets: (i) the *private* interfaces, and (ii) the *public* interfaces. Private interfaces define the services that are only accessed by the components of the architecture. Public interfaces define the services that may be also accessed by the application reusing the architecture. Figure 4.5 depicts all of the interfaces conformed by each architectural component.

The *Exception* component implements three public interfaces: (i) the interface IRaising, (ii) the interface IGetInformation, and (iii) the interface IUpdateInformation. The interface IRaising allows the application to raise exceptions by invoking the method raise. The interface IGetInformation makes some services available for the application and other architecture's components to obtain extra-information about the exception occurrences. Finally, the interface IUpdateInformation allows the application and the other components to update extra-information about exceptions.

The *Handler* component implements the private interface IInvocation. This interface allows the *Exception Handling Strategy* component to invoke an exception handler when this component has found an appropriate handler. The *Exception Handling Strategy* component conforms to the private interface ISearcher that provides the *Concurrent Exception Handling Action* component with the service for handler search. The *Concurrent Exception Handling Action* component implements the public interface ICooperation which provides the application with means of performing concurrent cooperative actions.

The components collaborate to realize the set of scenarios of the architecture. Figure 4.6 illustrates a scenario by means of a sequence diagram. This scenario shows the interactions between the application and the interfaces of the architecture's components

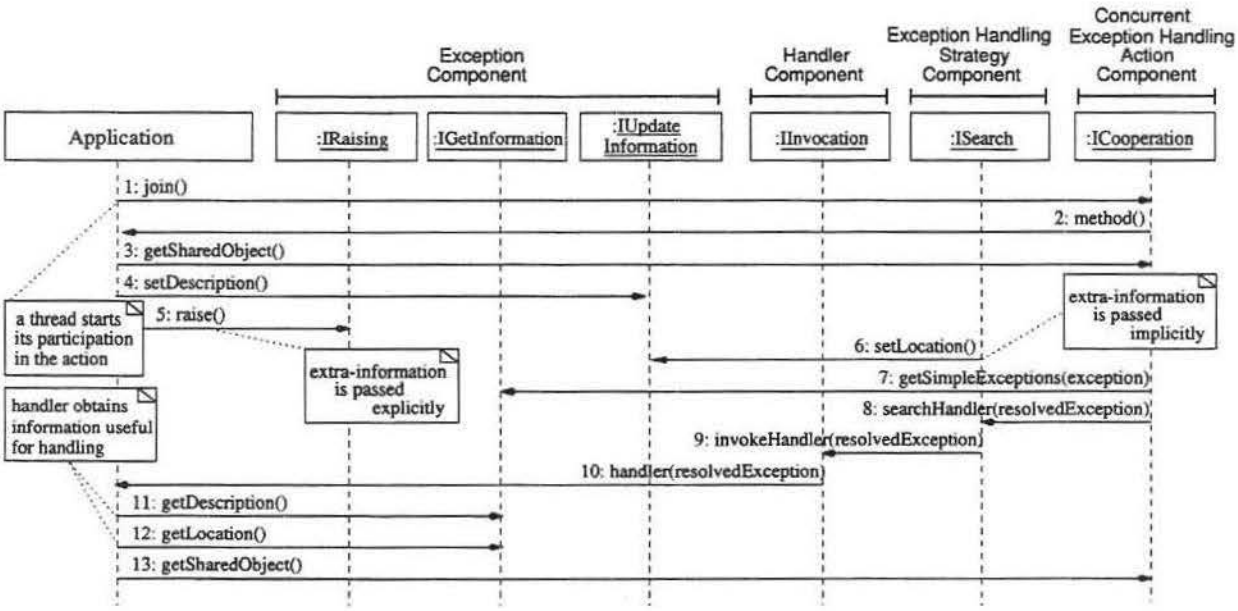


Figura 4.6: A Scenario of the Proposed Software Architecture.

in order to handle a cooperating exception that was raised during a concurrent cooperative action. A thread of the application invokes the method join in order to take part in the the action and perform a specific action participant (1). The *Concurrent Exception Handling Action* component invokes the application's method to be executed by the application's thread (2). While this method is being carried out by the participating thread, it obtains the shared objects used for inter-thread communication (3), and passes explicitly extra-information concerning the exception occurrence (4). During its execution, the application's method raises a cooperating exception (5).

After exception is raised, the architecture's components interact with each other to accomplish the management activities. Extra-information about that exception occurrence is updated implicitly by the components (6). The action participants are synchronized and exception resolution process is executed within the *Concurrent Exception Handling Action* component. During the resolution, this component communicates with the *Exception* component in order to obtain extra-information about the raised cooperating exceptions (7). The *Concurrent Exception Handling Action* component asks the *Exception Handling Strategy* component to search the handler for the resolved exception (8), and the later asks the *Handler* component to invoke it (9). The *Handler* component invokes the handler defined in the application (10). The handler obtains extra-information useful for handling the resolved exception (11-12), and shared objects that are used in the cooperation with the other handlers being executed concurrently (13).

4.4.3 The Architecture Refinement

Separation of Concerns. As stated previously, software designers tailor the components of the proposed architecture to add the functionality related to specific applications. Note that each architectural component may include application's functionality and management activities for exception handling. In order to obtain a clear separation of concerns between the application's functionality and the exception handling services, the architecture and their components incorporate a meta-level architecture, following the overall structure of the *Reflection* pattern (Section 4.3.1). Figure 4.7 presents the proposed meta-level architecture which is composed of two dimensions: the base level, and the meta-level. The architecture's base level encompasses the application-dependent elements, such as exceptions, handlers, normal activities, and concurrent cooperative actions. The architecture's meta-level consists of meta-objects which perform the management activities for exception handling.

Transparency. The *Reflection* pattern also captures the benefit of transparency obtained by means of computational reflection. For the purposes of this work, object states, results and invocations of methods of the application (base-level) are intercepted and reified by the MOP, and potentially checked and altered by the meta-objects (meta-level) in order to carry out the management activities for exception handling. For instance, results of methods are checked transparently by the meta-objects to verify if such methods have raised any exception. MOP intercepts at run-time the exceptional results and deviates the normal control flow of the base-level application to the exceptional one at the meta-level. When the management activities are finished, MOP returns the computation to the application's normal flow. Therefore, the meta-objects execute their management activities transparently from the viewpoint of the base-level.

Refinement of the Components and Design Patterns. While application designers reuse the architecture and refine its components to satisfy their needs, some problems arise in this context, such as: (i) how do they specify the simple and cooperating exceptions? and how do they do it uniformly?, (ii) how do they specify the handlers?, and (iii) how to execute the synchronization of the action participants and other management activities in a way that is transparent to the application?. In this work, design patterns are proposed in order to refine the general components of the proposed architecture, providing the detailed design solutions. The proposed design patterns present solutions for specific design problems of the corresponding components, and are used to describe the design and dynamics aspects of each architectural component.

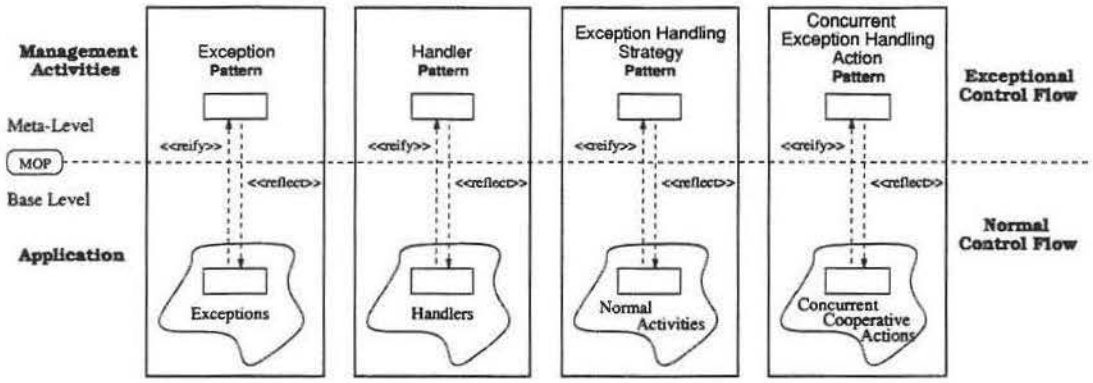


Figura 4.7: The Architecture Refinement.

4.5 Design Patterns for Exception Handling

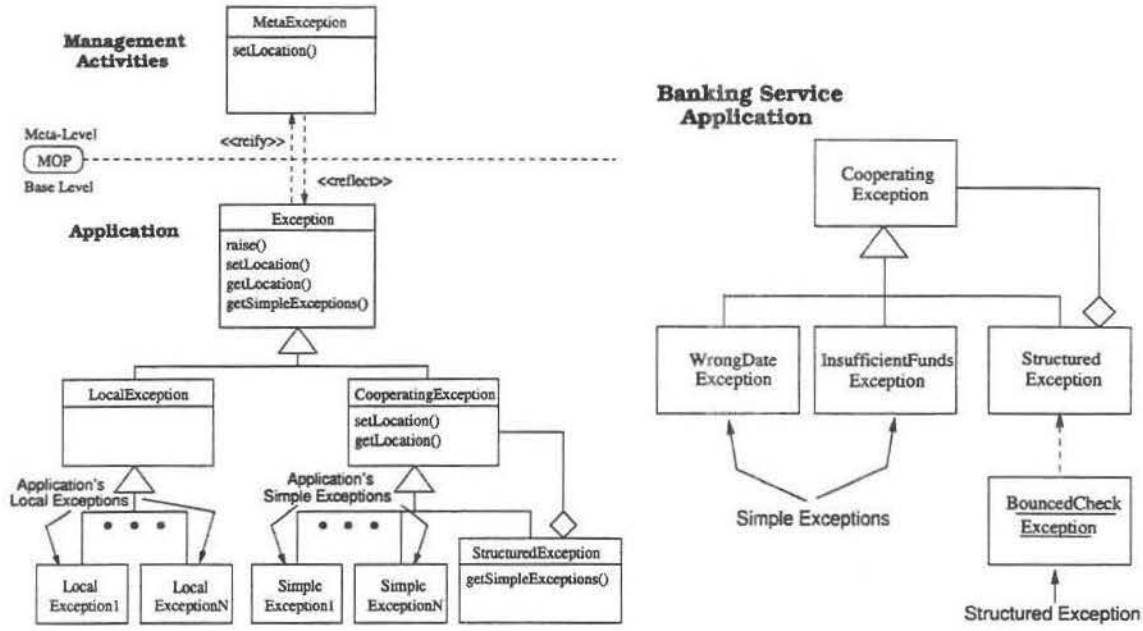
4.5.1 The *Exception* Pattern

Context. Software designers want to specify the local and cooperating exceptions of their applications. These exceptions may be raised at run-time during the application's normal activity. Extra-information is required by the application in order to handle an exception occurrence.

Problem. The software architecture should provide means by which the application developers define and raise the local and cooperating exceptions. Moreover, a flexible and reusable software architecture is required to make the exception specification easier and to separate concerns between application exceptions and extra-information management. Several *forces* are associated with this design problem:

- Local and cooperating exceptions should be defined uniformly.
- The effort of software designers to compose exception trees should be minimized.
- The exception occurrence itself should keep extra-information necessary for its handling.

Solution. Use the *Reflection* architectural pattern in order to separate classes responsible for managing extra-information (meta-level) from the ones used to specify application exceptions (base level). Different types of exceptions are organized hierarchically as classes which are termed *exception classes*. Exception occurrences are base-level objects created at run-time when an exception is raised, and are termed *exception objects*. Exceptions are raised by calling the method raise on exception objects. Meta-objects are associated transparently with exception objects for keeping extra-information about the



(a) General Structure (b) An Example

Figura 4.8: Class Diagram for the *Exception* Pattern.

exception occurrences. Extra-information is reified as meta-information. A meta-object keeps meta-information collected at run-time about the corresponding exception occurrence. Meta-objects alter transparently the state of the exception objects in order to make this information available for the application. As a result, the exception object keeps extra-information necessary for its handling. The application accesses this information by invoking methods on exception objects.

Structure. The *Exception* Pattern consists of exception classes, and meta-objects. Meta-objects of the type MetaException are associated with instances of base-level exception classes, i.e., meta-objects are associated with exception objects. Application developers are provided with three main exception classes – LocalException, CooperatingException and StructuredException (Figure 4.8(a)). These classes derive from the root class Exception. The class LocalException defines the local exceptions of applications; it is subclassed by application designers in order to specify the local exceptions. Exception trees are easily specified – an application developer only needs to create a class for each simple exception by subclassing the class CooperatingException, and a new instance of the class StructuredException for each structured exception. An exception object of the type StructuredException stores the simple and/or structured exceptions which compose it. The method getSimpleExceptions returns the simple exceptions that compose a structured exception.

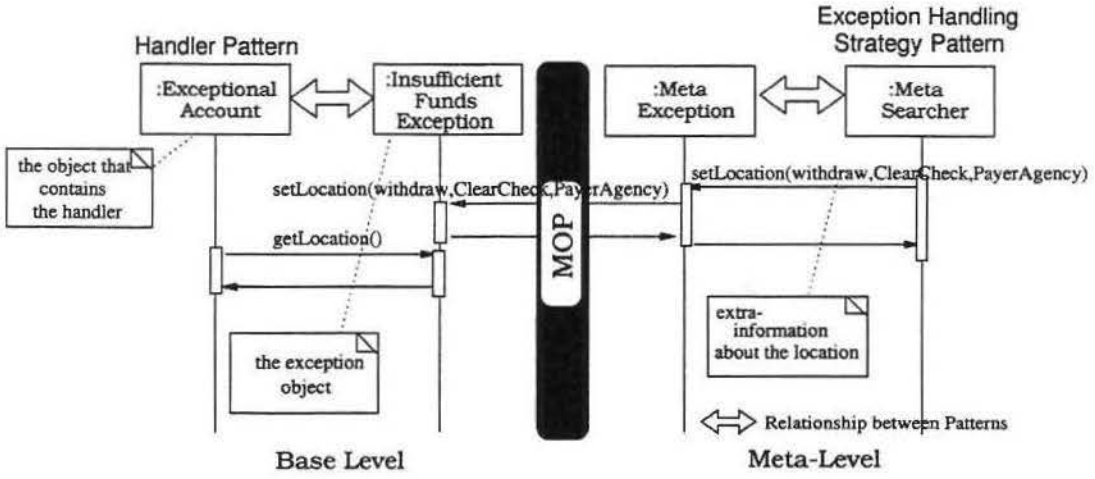


Figura 4.9: Interaction Diagram for the *Exception* Pattern.

Figure 4.8(b) shows an instance of the proposed pattern for defining an exception tree in the banking service application (Section 4.2.2). This scheme for definition of exception trees is similar to the structure of the *Composite* design pattern [20].

Dynamics. Figure 4.9 presents the interaction diagram that illustrates a scenario for the banking service example (Section 4.2.2). A meta-object (of the type MetaSearcher), associated with the application object that has raised the exception InsufficientFundsException, reifies extra-information about the location where the exception was detected. The information includes the method, the action and the action participant where the exception was raised. This meta-object sends the extra-information to the meta-object associated with the application's exception object that represents that exception occurrence. The meta-object updates extra-information about the exception occurrence by invoking transparently the method setLocation on the application's exception object. The invocation and the update are transparent from the viewpoint of the application. Then the method getLocation is invoked by the application handler in order to receive the extra-information related to the location.

Known Uses. The representation of exceptions as classes is a design solution adopted by several systems and programming languages, such as Java, C++ and Arche.

Consequences. The *Exception* Pattern offers the following benefits:

- *Uniformity.* Both local and cooperating exceptions are uniformly defined as classes. Moreover, the *Exception* pattern adopts the *Composite* pattern to define exception trees. As a result, it allows application designers to treat simple exceptions and its compositions (structured exceptions) uniformly.

- *Simple to Use.* Exception trees are easily defined. The proposed base-level classes

allow application developers to define exception trees without writing an exception resolution procedure for each concurrent cooperative action of the application. The exception resolution process is performed transparently by the meta-level (Section 4.5.4).

- *Reusability and Extendibility.* The representation of local and cooperating exceptions as classes promotes the reusability and extendibility of the exception classes. In addition, the separation of concerns provided by the *Exception* pattern also promotes the reusability of the management services.

- *Readability and Maintainability.* Applications whose exceptions are represented as objects are easier to understand and maintain than applications where exceptions are simply symbols (numbers or strings) [25].

- *Easy incorporation of default handlers.* Since exceptions are represented as classes, default handlers can be defined as methods on exception classes. In case the application developers have not defined more specific handlers, the handler method on the exception class can be activated by the meta-level.

4.5.2 The *Handler* Pattern

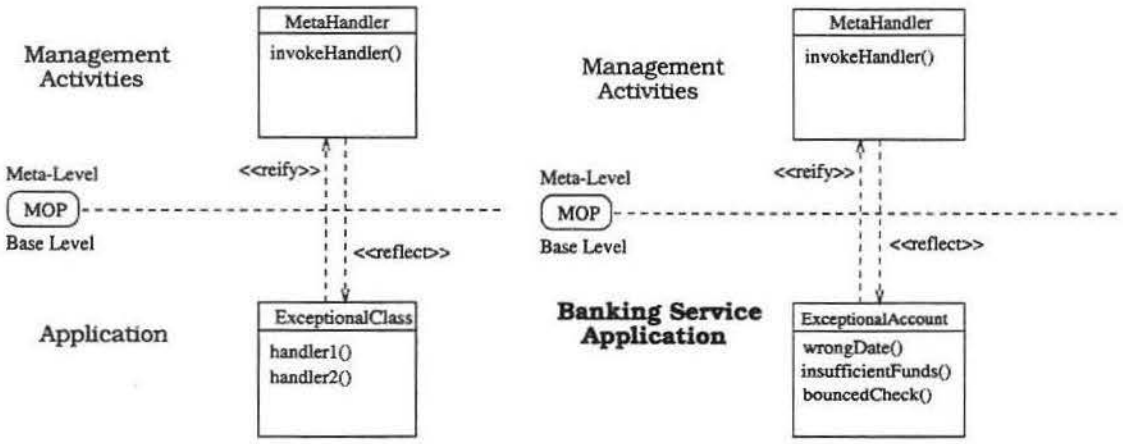
Context. Software designers want to specify the handlers for the local and cooperating exceptions that are expected to occur during the normal activity of their applications. A handler is invoked when the corresponding exception is raised.

Problem. The infra-structure of the software architecture should be organized in order to allow application developers to define the exception handlers in a way that separates them from the application's normal activity. In addition, this infra-structure should promote the separation between the application components containing the exception handlers and the architectural components responsible for invoking the eligible handler. The following *forces* shape the solution:

- Exception handlers for local and cooperating exceptions should be defined in an uniform manner.

- The software architecture should include multi-level attachment of handlers (Section 4.2.1).

Solution. Use the *Reflection* architectural pattern in order to separate the class responsible for invoking handlers (meta-level) from the classes used to specify the application handlers (base level). The base-level defines the application classes that implement the



(a) General Structure (b) An Example

Figura 4.10: Class Diagram for the *Handler* Pattern.

handlers for local and cooperating exceptions. The meta-level consist of meta-objects responsible for invoking the handlers.

Structure. The *Handler* Pattern consists of two kinds of elements: (i) exceptional classes, and (ii) meta-objects of the type MetaHandler (Figure 4.10(a)). The *exceptional classes* are located at the base-level and define the error handling activities of a specific application. The methods of exceptional classes are the handlers for the local and cooperating exceptions raised during the execution of normal classes' methods. The *normal classes* are located at the base-level and implement the application's normal activities (see Section 4.5.3). Therefore, exceptional classes implement the handlers of the application and they are attached to the corresponding normal classes. Meta-objects of the type MetaHandler are associated with exceptional classes, and are responsible for invoking transparently the exception handlers.

Exceptional classes can contain handlers attached to classes, objects and methods. Each exceptional class may contain handlers for coping with the local and cooperating exceptions; they are invoked when these exceptions are raised during the execution of methods of the corresponding normal class. Figure 4.10(b) shows an instance of this pattern for the banking service example. The methods of the ExceptionalAccount are the handlers for the simple and structured exceptions that can be raised while the methods of the corresponding normal class (the class NormalAccount – Section 4.5.3) are being executed during a concurrent cooperative action.

Dynamics. Suppose the method withdraw is being executed concurrently during a concurrent cooperative action and raises the exception InsufficientFundsException; another

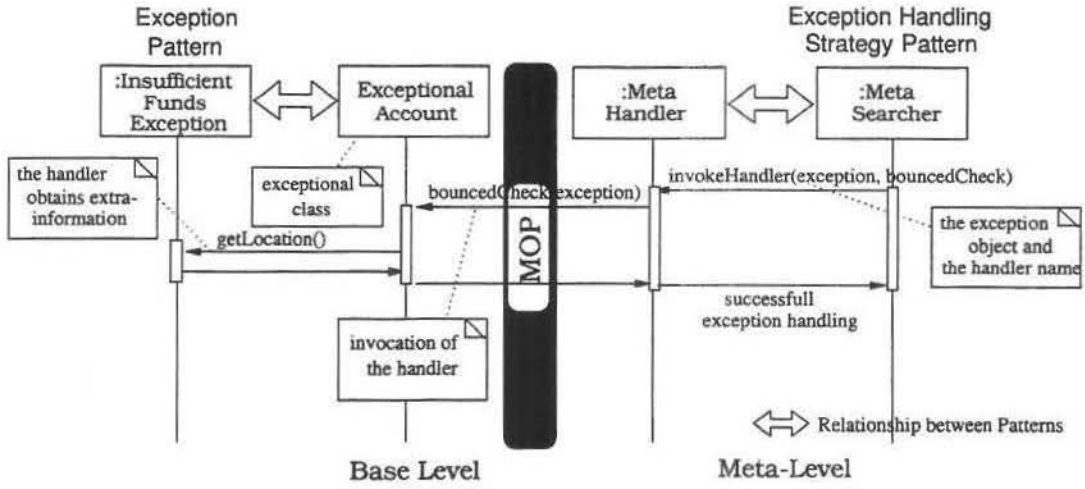


Figura 4.11: Interaction Diagram for the *Handler* Pattern.

method is being executed concurrently during this concurrent cooperative action and also raises an exception, the exception WrongDateException. The concurrent raising of these simple exceptions means the occurrence of the structured exception BouncedCheckException, and the subsequent invocation of the handlers to deal with this structured exception. Figure 4.11 illustrates the transparent invocation of the appropriate handler by the meta-object associated with the exceptional class. During the execution of the handler, it gets extra-information about the location where the exception InsufficientFundsException was raised.

Known Uses. The work [31] also uses the computational reflection technique in order to obtain meta-information about the application and invoke the suitable handler when an exception is raised. Meta-level structures implement the exception handling mechanism while at the base level resides the application. Finally, handlers also are implemented as ordinary methods. The approach presented in [48] uses a variant of the *Handler* pattern. This variant transfers the handler methods from the exceptional classes to the meta-level. The meta-objects associated with the normal classes contain application's methods responsible for performing the exception handling. Instead of utilizing reflective principles to complete the separation between application and management mechanisms, this variant explores reflection to separate normal and exceptional code of the application.

Consequences. The *Handler* Pattern offers the following consequences:

- *Uniformity.* Handlers for both local and cooperating exceptions are defined uniformly as methods of exceptional classes.
- *Readability and Maintainability.* The pattern provides explicit separation between normal and error-handling activities, which in turn promotes readability and

maintainability.

- *Flexibility.* The multi-level attachment of handlers allows developers to attach handlers to the respective levels of classes, objects and methods.
- *Reusability.* The use of normal and exceptional classes allows application designers to compose an *exceptional class hierarchy* that is orthogonal to the normal class hierarchy of the application. The exceptional classes are organized hierarchically so that resultant hierarchy is orthogonal to the *normal class hierarchy.* Exceptional class hierarchies allow exceptional subclasses inherit handlers from their superclasses and, consequently, they allow exceptional code reuse. When reuse is not desired, the handler method can be redefined at the subclasses.
- *Minor loss in writeability.* A protected region can not be defined as a statement.
- *Lack of Static Checking.* A possible disadvantage of this pattern is that may not be easy to check statically if handlers have been defined for all specified exceptions. However, alternative solutions may be applied (Section 4.6).

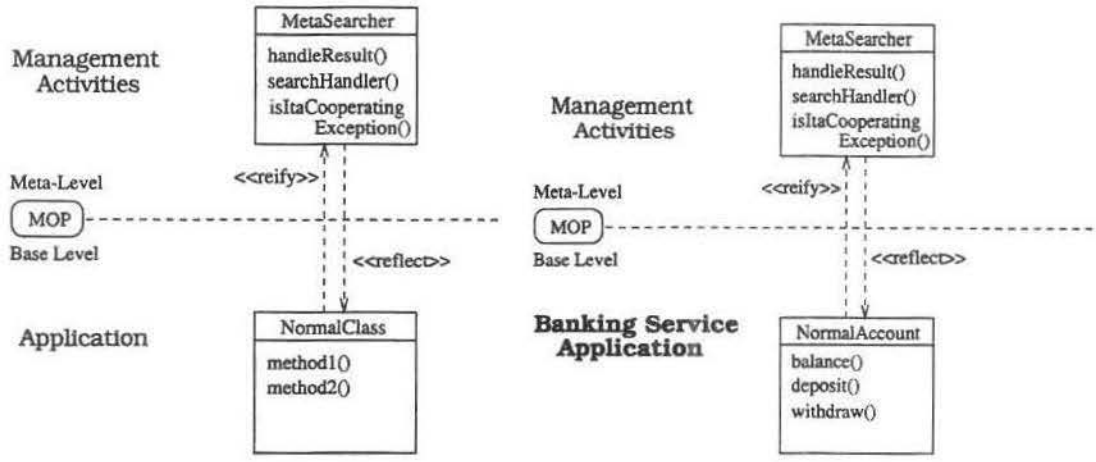
4.5.3 The *Exception Handling Strategy* Pattern

Context. Exception occurrences can be detected during execution of a protected region of the application's normal activity. The normal control flow is deviated to the exceptional one and an appropriate handler is searched.

Problem. The software architecture should be organized in a disciplined manner: the components responsible for the deviation of the normal control flow and for the handler search should perform their management activities in a non-intrusive way to the application. The following *force* arises when dealing with such a problem:

- The chosen model for continuation of the control flow should be termination since it is more suitable for developing dependable systems (Section 4.2.1).

Solution. Use the *Reflection* architectural pattern in order to separate classes responsible for the management activities (meta-level) from the ones that implement the normal activities of the application (base level). The base-level defines the application's logic where normal classes implement the normal activities. The meta-level consists of meta-objects which search transparently for the exception handlers. Meta-objects are associated with instances of the normal classes, and maintain meta-information concerning the protected regions defined at the base-level. A protected region can be a method, an object, and a class. MOP itself is responsible for intercepting method results and changing the normal



(a) General Structure (b) An Example

Figura 4.12: Class Diagram of the *Exception Handling Strategy* Pattern.

control flow to the exceptional one when exceptions are detected by transferring control to the meta-level. With the available meta-information, meta-objects find the handler that should be executed when an exception occurrence is detected in a given protected region. When the execution of the handlers is finished successfully, MOP returns the control flow to the application's normal computation according to the termination model.

Structure. The *Exception Handling Strategy* Pattern introduces two types of elements: (i) normal classes, and (ii) meta-objects of the type MetaSearcher (Figure 4.12(a)). The *normal classes* are located at the base-level and define the normal activities of a specific application. They are attached to the corresponding exceptional classes. Figure 4.12(b)) pictures an instance of this pattern for the banking service application. This figure shows the normal class NormalAccount; it is attached to the exceptional class ExceptionalAccount (Section 4.5.2). Meta-objects of the type MetaSearcher are associated with instances of normal classes, and are responsible for the interruption of the normal control flow and the handler search.

Dynamics. Figure 4.13 presents the interaction diagram for the banking service example. The method withdraw is being performed concurrently during a concurrent cooperative action. The exception InsufficientFundsException is returned as the result of withdraw since it has raised this exception during its execution. MOP intercepts and reifies the result of withdraw, and notifies the meta-object about the exceptional result by means of the method handleResult. The meta-object checks if the exception occurrence is a local or a cooperation exception. If it is a local exception, the meta-object searches immediately for the

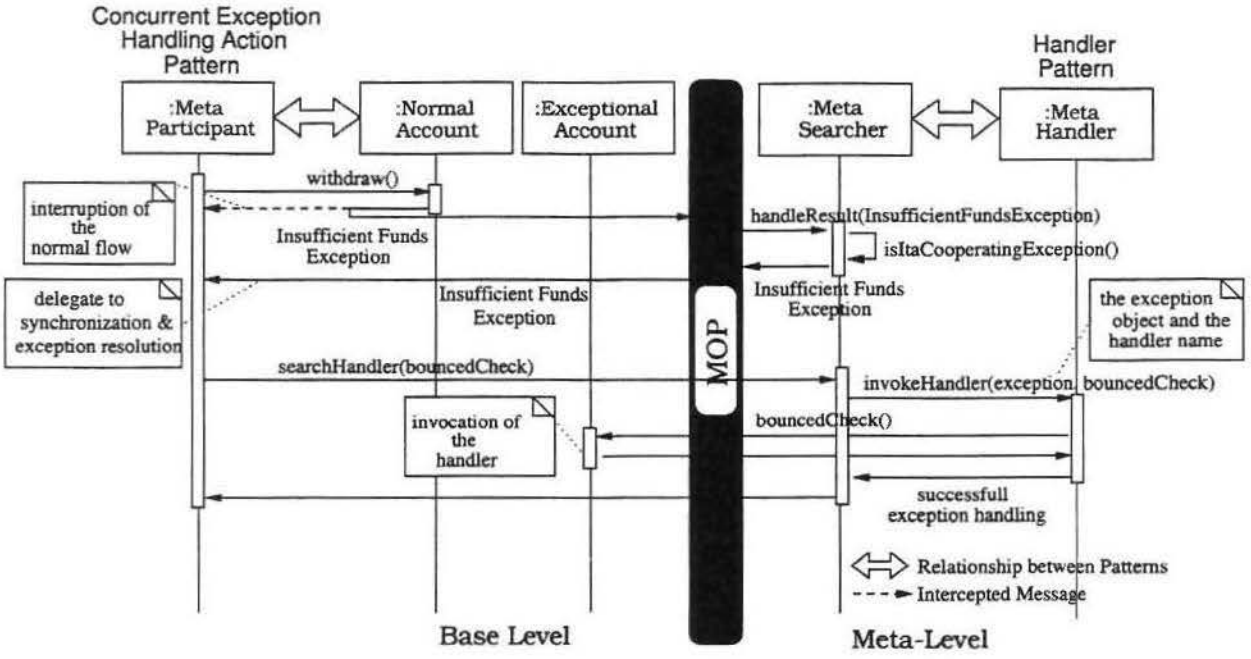


Figura 4.13: Interaction Diagram for the *Exception Handling Strategy* Pattern.

exception handler based on the available meta-information. Otherwise, the cooperating exception is firstly delegated to the meta-object responsible for the participant synchronization and exception resolution. InsufficientFundsException is then delegated since it is a cooperating exception. After exception resolution is accomplished, the meta-object is required to find a handler for the resolved exception, the exception BouncedCheckException. The handler is found and the invocation of it is delegated to the appropriate meta-object (Section 4.5.2). Since the exception handler is executed successfully, the control is passed to the meta-object responsible for the participant synchronization, which in turn will deviate the exceptional control flow to the normal one.

Known Uses. The work [48] presents a variant of the *Exception Handling Strategy* pattern. In this variant, the exception itself is the reified entity instead of a method result. This alternative design solution allows the exception itself to control the handling. Consequently, it is possible to implement the resumption model since the control flow is stopped exactly at the point of the exception raising. The work [31] uses the reflection technique in order to obtain at compile-time information concerning protected regions and the handlers that are attached to them.

Consequences. The *Exception Handling Strategy* Pattern offers the following consequences:

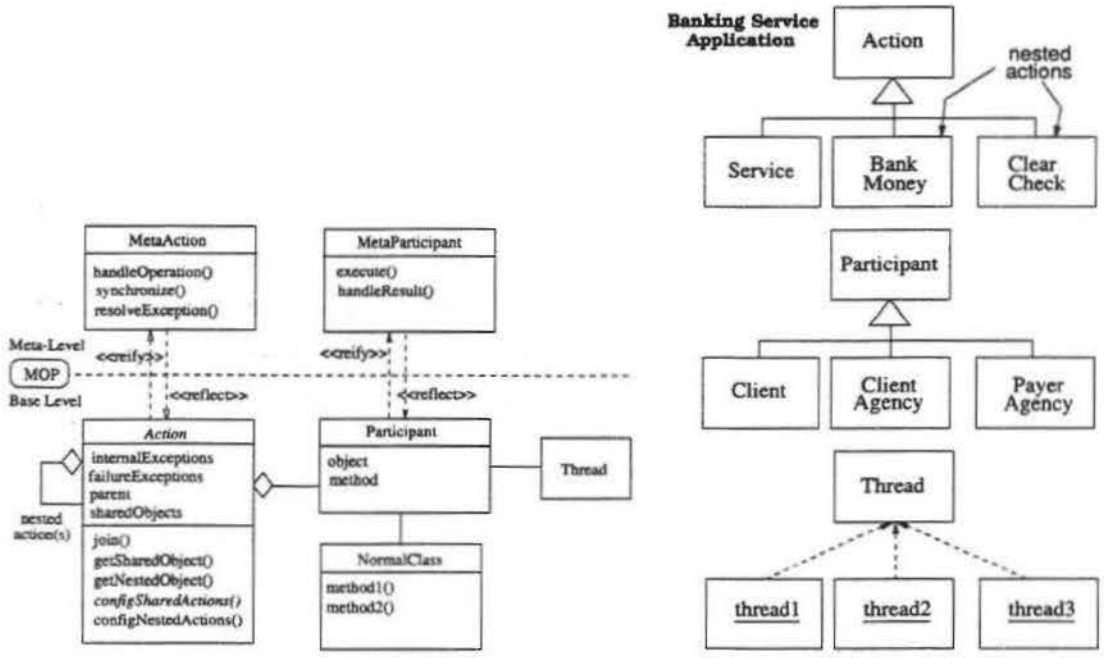
- *Transparency.* The meta-level objects bind transparently the normal activity and corresponding handlers without requiring from programmers the use of new keywords to specify protected regions.
- *Readability and Maintainability.* The normal code is not amalgamated with the exceptional code. As a consequence, both normal and exception code are easier to read and maintain.
- *Compatibility.* The *Exception Handling Strategy* pattern can be used together with an exception handling strategy implemented in the underlying programming language, and they can complement each other.

4.5.4 The *Concurrent Exception Handling Action* Pattern

Context. Software designers want to specify concurrent cooperative actions. These actions must be controlled at run-time and their participants have to leave the action synchronously. During the execution of an action, a number of cooperating exceptions can be raised. As a consequence, a service of exception resolution is necessary to agree on the cooperating exception to be handled by all participants of the action.

Problem. The software architecture should provide means by which the software developers define the concurrent cooperative actions of their applications. Moreover, an disciplined and flexible approach is required to separate concerns and minimize dependencies between the concurrent cooperative actions of the application and the strategy for concurrent exception handling (i.e. the management mechanisms for synchronization and exception resolution). Several *forces* are associated with this design problem:

- The definition of concurrent cooperative actions should be done in a structured manner to avoid an increase in the software's complexity.
- The inter-thread communication should use shared objects (Section 4.2.2).
- The strategy for concurrent exception handling should be a consistent extension of the general strategy for exception handling.
- The blocking approach should be used for concurrent exception handling since it simpler and easier to implement (Section 4.2.2).



(a) General Structure (b) An Example

Figura 4.14: Class Diagram of the *Concurrent Exception Handling Action* Pattern.

Solution. Use the *Reflection* architectural pattern for segregating classes responsible for the management mechanisms (meta-level) from the classes which must be derived for defining the concurrent cooperative actions of the application (base-level). Based on a meta-level architecture, the *Concurrent Exception Handling Action* pattern separates objects into well-defined levels. The base-level provides developers with classes for creating the concurrent cooperative actions of their applications; the definition of nested actions is also supported in order to control the system's complexity and allow better organization of both normal and error handling activities of the enclosing action. MOP itself intercepts and reifies invocations of methods and their results. The meta-level implements the management mechanisms based on reified invocations and results, and on the available meta-information.

Structure. The *Concurrent Exception Handling Action* Pattern introduces five types of objects: (i) Action, (ii) Participant, (iii) Thread, (iv) MetaParticipant, and (v) MetaAction (Figure 4.14(a)). The class Thread represents the threads which intend to participate in a concurrent cooperative action. Developers create their threads, and extend the classes Action and Participant by subclassing them to implement their concurrent cooperative actions. Instances of these subclasses represent at run-time a specific action and their participants respectively. Developers should redefine the method ConfigureSharedObject while subclassing the class Action. The method ConfigSharedObject implements the application-dependent activity which consists of creating shared objects used for purpose of inter-participant communication (inter-thread communication). In order to access these objects, each participant have to ask to its corresponding action references to these objects by means of the method getSharedObject. If an action is composed of one or more nested actions, developers should also redefine the method ConfigNestedActions in order to create the objects that represent the nested actions. In order to access these objects, each participant have to ask to its corresponding action references to these objects by invoking the method getNestedAction. Each object of the type Participant holds references to: (i) its action, and (ii) an object and its method that will be executed during the action by a thread. Instances of the class Action have references to: (i) action participants, (ii) internal and failure exceptions, (iii) its parent (enclosing action), (iv) its nested actions, and (v) shared objects. Internal exceptions are the exceptions that should be handled within action by all action participants, while external exceptions are the exceptions that should be signaled to the enclosing action. Figure 4.14(b) shows an instance of the proposed pattern for defining the actions, the participants, and the threads for the banking service application.

Instances of the class MetaParticipant are associated with instances of subclasses of Participant. These meta-objects are responsible for: (i) execute the application's method which is held by its associated participant, (ii) inform to its corresponding MetaAction

about the end of this method execution, and (iii) ask the appropriate meta-object to invoke the handler associated with a resolved exception. Instances of the class MetaAction are associated with instances of subclasses of Action. These meta-objects are responsible for: (i) perform the exception resolution, and (ii) synchronize the action participants.

Dynamics. Figure 4.15 presents the interaction diagram for the banking service example. The diagram illustrates the application's thread performing the participant PayerAgency within the action ClearCheck. This thread intending to participate in the action, calls the method join on the object ClearCheck corresponding to that action. The thread informs to the action what participant (the participant PayerAgency) it intends to execute during the concurrent cooperative action. MOP intercepts and reifies the invocation of join, and notifies the meta-object MetaAction about this invocation by means of the method handleOperation. This meta-object checks to see if is allowed to play that participant in this action, and if so, the meta-object MetaParticipant executes the method withdraw that is attached to that participant. While this method is being carried out by the thread, it obtains the shared objects used for inter-thread communication. The exception InsufficientFundsException is returned as the result of withdraw since it has raised this exception during its execution (Section 4.5.2). MOP reifies the result of withdraw, and notifies the meta-object MetaParticipant about the exceptional result by means of the method handleResult. The action participants are synchronized and exception resolution process is accomplished based on the available meta-information. For instance, the meta-object MetaAction communicates with the meta-object responsible for maintain extra-information about the raised cooperating exceptions (Section 4.5.1). The meta-object MetaParticipant receives the resolved exception and then asks the eligible meta-object to search the handler for the resolved exception. Note that after the thread asks to start its activity within the action, all management activities are performed by the meta-level in a way that is transparent to the application.

Known Uses. The work [61] proposes a non-reflective and distributed variant of this pattern. This works proposes an algorithm for concurrent exception handling in distributed object systems. Exception resolution and the final synchronization is performed in a distributed way, and the information concerning the action must be held by each participant. Each participant must keep a copy of the algorithm and the management is performed by means of message exchange. However, the class Action is not necessary. Zorzo's CAAction framework [74] also provides software developers with a number of classes to structure their concurrent applications. However, it uses a non-reflective variant of the *Concurrent Exception Handling Action* Pattern. Programmers extend two classes of the framework in order to implement their concurrent cooperative actions. Both classes are similar to the classes Action and Participant regarding their responsibilities.

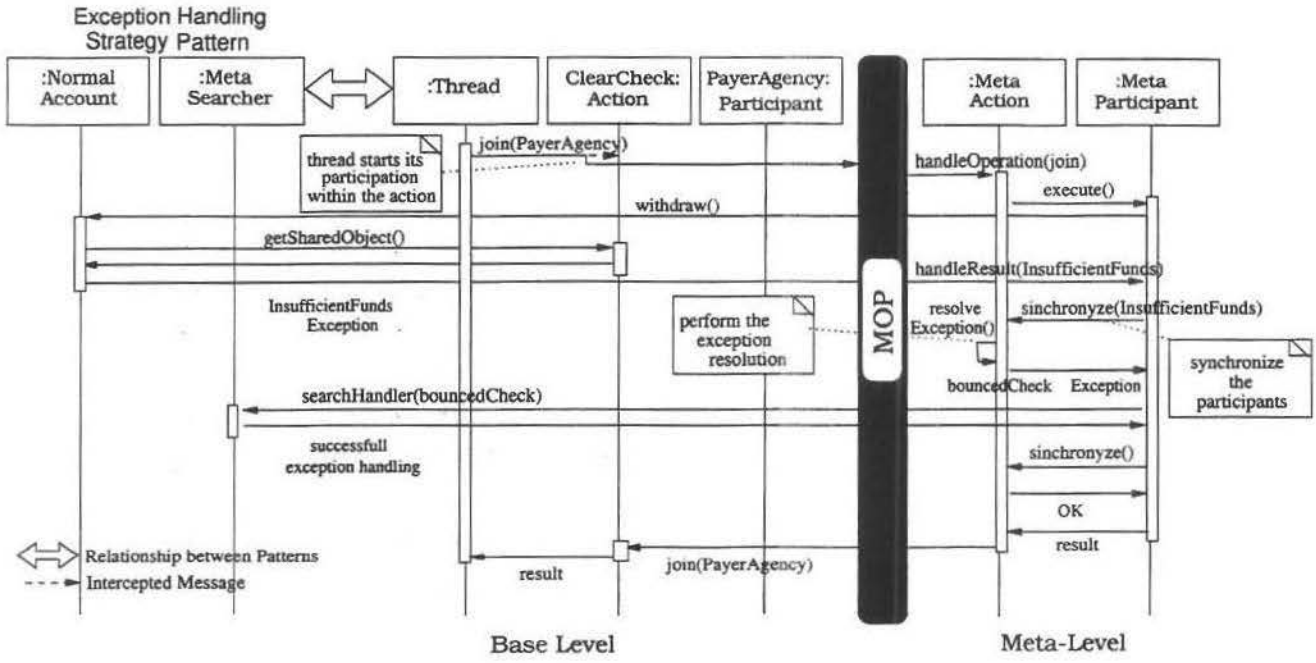


Figura 4.15: Interaction Diagram for the *Concurrent Exception Handling Action* Pattern.

Consequences. The *Concurrent Exception Handling Action* Pattern offers the following consequences:

- *Uniformity.* The strategy for concurrent exception handling is a consistent extension of the general strategy for exception handling.
- *Transparency and Simple to Use.* Management mechanisms for exception handling are performed transparently to the application. Programmers fix their attention on definition of concurrent cooperative actions, which is an application-dependent issue.
- *Complexity Control.* The pattern allows programmers to define nested actions.
- *Readability, Reusability and Maintainability.* The application code is not intermingled with invocations of methods responsible for synchronization and exception resolution. As a consequence, it improves readability, which in turn improves reusability and maintainability.
- *Minor loss in efficiency due to the blocking model.* In pre-emptive schemes, there is inherently no wasted time but the feature required, namely pre-emptive thread interruption, is not readily available in many systems and programming languages. However, mechanisms such as timeouts and run-time error checks can increase the efficiency of blocking schemes and decrease the amount of time wasted by allowing early detection of either the error or the abnormal behavior of the participant that raised the exception and is waiting for the other participants.

4.6 Implementation Issues

We have implemented the proposed software architecture using the Java programming language without any changes to the language itself by means of a meta-object protocol called *Guaraná* [51]. 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 by means of composers. The proposed software architecture has allowed that the meta-objects of our exception handling system be integrated with meta-objects responsible for other quality attributes, such as persistency and security.

The proposed software architecture can be implemented using any language as long as the environment supports computational reflection. The proposed architecture also can be implemented using languages where ever exists an exception handling mechanism without

conflict with the existing one. A disadvantage of our approach is that may not be easy to perform checks statically (Section 4.5.3). However there are some features which can help programmers to avoid mistakes: post- and pre-processors, libraries, syntax-oriented editors and macro-processing. In addition, the mechanism of computational reflection may facilitate checks performed before program execution by obtaining information about the application using the exception handling mechanism.

4.7 Related Work

Even though many object-oriented programming languages include exception handling facilities, only the Arche language [33] provides actual support for concurrent exception handling. The exception handling mechanism of this language allows user-defined resolution of multiple exceptions amongst a group of objects that belong to different implementations of a given type; however, this approach is not generally applicable to the concurrent exception handling of multiple interacting objects of different types.

The paper [58] describes a scheme for concurrent exception handling based on atomic action structures for the Ada95 language. In this approach, application programmers have to implement a exception resolution function for each concurrent cooperative action. Programmers are responsible for deciding how implement this resolution function. Application code is also intermingled with invocations of *head processes* which are procedures for synchronizing the participants while exiting from the action. In this way, application objects are polluted with explicit references for *head processes*.

The coordinated atomic action concept [69] was introduced as a unified approach for structuring complex concurrent activities and supporting error handling between multiple interacting objects in a concurrent object-oriented system. Zorzo *et al.* [74] had developed an object-oriented scheme for implementing coordinated atomic actions. In this scheme, application programmers are provided with a number of classes to structure their application. However, such solution proposes a very simple exception handling mechanism. There is a single method intended to handle the cooperating exceptions raised during the cooperative activity. The structure of such a handler is very complex since a single handler must incorporate handling measures for all cooperating exceptions.

The work of Hof *et al.* [31] describes an approach for exception handling based on meta-programming and computational reflection. Their implementation was carried out in a specific system but it also could be implemented to most other systems that support meta-programming. However, this approach does not support concurrent exception handling and its exception handling model is not object-oriented.

4.8 Conclusions and Ongoing Work

In recent years exception handling mechanisms have become a important part of mainstream object-oriented programming languages. However, designers of these mechanisms have not paid enough attention to concurrent exception handling. Recently, a number of different approaches have been proposed to the provision of concurrent exception handling. However, these proposals are introduced in an 'ad hoc' way and are not integrated uniformly with the exception handling strategy for sequential programs. These current solutions make the task of application developers very difficult since they often are responsible for implementing various management activities. In fact, such solutions pollutes application code with explicit references and invocations of procedures for exception resolution and thread synchronization. Consequently, the use of these mechanisms reduces the demanding quality requirements arising with modern software systems, such as readability, maintainability and reusability.

This paper presents a generic software architecture to introduce exception handling into dependable object-oriented software. The proposed architecture supports uniformly concurrent and sequential exception handling, and may be implemented without create any linguistic construction as a result for the underlying language. Our architecture provides during the first design stage the context in which more detailed design decisions are made in later design stages related to exception handling. In this sense, this work also presents a set of design patterns which are used to describe the structural and dynamic aspects of the components of the proposed architecture. The design patterns incorporate well-proved solutions and their micro-architecture achieves a clear and transparent separation of concerns between the application's functionality and the exception handling services. Specific applications reuse the exception handling facilities provided by the proposed architecture's components, and the developers concentrate their attention on the application-dependent functionality.

Nowadays the component programming 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 dependable object-oriented software from the scratch. 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.

4.9 Resumo do Capítulo 4

Este capítulo apresentou um artigo que aborda a proposta de uma arquitetura de software genérica para mecanismos de exceções. A arquitetura proposta define os componentes arquiteturais de um mecanismo de exceções e a interação entre estes componentes. A arquitetura especialmente define um componente para tratamento de exceções concorrentes que é integrado uniformemente com os outros componentes arquiteturais. A arquitetura proposta é baseada em um mecanismo de reflexão computacional, sendo assim composta de dois níveis distintos. A divisão em níveis permite obter uma divisão clara entre as funcionalidades da aplicação e os serviços do mecanismo de exceções proposto. Consequentemente, a divisão obtida contribui efetivamente para a construção de um mecanismo simples e não intrusivo.

Os padrões de projeto propostos constituem soluções de projeto para problemas recorrentes no contexto de mecanismos de exceções. Estes padrões identificam soluções existentes e bem provadas, e concomitamente são aplicados para documentar os componentes da arquitetura de software proposta. A proposta do conjunto de padrões contribui ainda mais para a reutilização da arquitetura proposta no projeto de um mecanismo de exceções.

O próximo capítulo resume as conclusões do nosso trabalho, apresentando as principais contribuições e os possíveis trabalhos futuros.

Capítulo 5

Conclusão Geral

Esta dissertação concentrou-se no projeto e implementação de um mecanismo de tratamento de exceções para construção de software orientado a objetos confiável. Para desenvolvimento do mecanismo proposto, utilizamos técnicas avançadas de estruturação de software, tais como reflexão computacional e padrões de projeto. Durante o desenvolvimento deste trabalho, chegamos a vários resultados que formam as nossas principais contribuições:

1. Um estudo comparativo dos diferentes modelos de tratamento de exceções implementados em diversas linguagens orientadas a objetos e proposta de uma taxonomia que permite avaliá-los.

2. Proposta de um critério de projeto com os requisitos desejáveis para mecanismos de tratamento de exceções que serão utilizados na construção de sistemas orientados a objetos confiáveis. Um modelo ideal de tratamento de exceções é proposto tendo como base o critério de projeto definido. O modelo proposto especialmente dá suporte a tratamento de exceções concorrentes.

3. Projeto e implementação de um mecanismo de exceções para a linguagem Java utilizando a arquitetura de software reflexiva do Guaraná. O mecanismo implementa o modelo proposto de tratamento de exceções que contempla o critério de projeto definido.

4. Definição de uma arquitetura reflexiva genérica para o projeto de mecanismos de exceções que serão utilizados na construção de sistemas orientados a objetos confiáveis.

5. Proposta de um conjunto coeso de quatro padrões de projeto que documentam os aspectos estruturais e comportamentais dos componentes arquiteturais de um

mecanismo de tratamento de exceções, e incorporam boas soluções conhecidas para os problemas comuns no domínio desses mecanismos.

Duas contribuições são consideradas principais. A primeira delas, caracterizada por aspectos técnicos e usos práticos, é o projeto e implementação de um mecanismo de exceções que especialmente oferece suporte a tratamento de exceções concorrentes. A outra, caracterizada por aspectos abstratos e abordagem inovadora, é a definição de uma arquitetura de software genérica e o conjunto padrões de projeto relacionados que permitem a construção de mecanismos de exceções em diferentes linguagens de programação.

As aplicações das idéias apresentadas nesta dissertação são as seguintes:

- Auxílio a desenvolvedores de aplicações orientadas a objetos confiáveis, através do critério e modelo propostos, na escolha de um mecanismo de exceções adequado para a construção das suas aplicações.
- Auxílio a engenheiros de software e/ou projetistas de linguagens de programação no desenvolvimento de mecanismos de tratamento de exceções.
- Adição pouco intrusiva de um mecanismo de exceção a aplicações confiáveis existentes, através da arquitetura de software e padrões de projeto propostos.

As principais linhas de pesquisa que podem ser seguidas a partir do nosso trabalho são:

- Tradicionalmente, desenvolvedores de sistemas orientados a objetos postergam a preocupação com tratamento de exceções para as fases posteriores de projeto e implementação. Melhores resultados poderiam ser obtidos se as situações excepcionais fossem consideradas desde a fase de análise. Nesse contexto, faz-se necessário o desenvolvimento de uma abordagem para construção de software orientado a objetos, onde as atividades de tratamento de erros sejam incorporadas de forma disciplinada durante as fases de análise, projeto e implementação.
- O esforço de projetistas de sistemas orientados a objetos confiáveis deveria ser minimizado enquanto utilizando um mecanismo de exceções. Ferramentas CASE contribuem decisivamente na construção de sistemas complexos. Nesse sentido, a implementação de ferramenta CASE para definição de exceções, dos comportamentos normais e excepcionais pode auxiliar projetistas de aplicações confiáveis durante o processo de desenvolvimento.
- Diferentes tipos de aplicações requerem diferentes requisitos de um mecanismo de exceções. Um *framework* orientado a objetos [34] é um sistema de software que

incorpora uma arquitetura flexível e pode ser extendido para produzir aplicações com diferentes requisitos. A nossa arquitetura de software proposta pode ser utilizada e extendida para a implementação de um *framework* de tratamento de exceções. Os pontos flexíveis do *framework* poderiam ser configurados de acordo com os requisitos da aplicação utilizando o *framework* de tratamento de exceções.

Bibliografia

[1] R. Balter, S. Lacourte, and M. Riveill. The Guide Language. *The Computer Journal*, 7(6):519–530, 1994.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison-Wesley, Massachusetts, USA, 1998.

[3] G. Booch. *The Unified Modeling Language User Guide.* Addison Wesley Publishing Company, MA, 1998.

[4] Borland. *Object Pascal Language Guide – Borland Delphi for Windows 95 & Windowns NT - Version 2.0*, 1996.

[5] Borland. *User's Guide – Borland Delphi for Windows 95 & Windowns NT - Version 2.0*, 1996.

[6] A. Burns and A. Wellings. *Real-Time Systems and Their Programming Languages.* Addison-Wesley, 1996.

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

[8] J. Purchase C. Dony and R. Winder. Exception Handling in Object-Oriented Systems. In *ECOOP'91*, pages 17–30, 1991. Report on ECOOP'91 Workshop W4.

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

[10] F. Cristian. Exception Handling and Software Fault Tolerance. *IEEE Transactions on Computers*, C-31(6):531–540, June 1982.

[11] F. Cristian. Exception Handling. Technical Report RJ5724, IBM, 1987.

[12] F. Cristian. Exception handling. In T. Anderson, editor, *Dependability of Resilient Computers*, pages 68–97. Blackwell Scientific Publications, 1989.

[13] Q. Cui and J. Gannon. Data-Oriented Exception Handling. *IEEE Transactions on Software Engineering*, 18(5):393–401, May 1992.

[14] R. de Lemos and A. Romanovsky. Exception Handling in a Cooperative Object-Oriented Approach. In *2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, France, May 1999.

[15] C. Dony. An Object-Oriented Exception Handling System for an Object-Oriented Language. *Lectures Notes in Computer Science*, 322:146–161, August 1988.

[16] C. Dony. Exception Handling and Object-Oriented Programming: Towards a Synthesys. *Sigplan Notices*, 25(10):322–330, October 1990.

[17] S. Drew and K. Gough. Exception Handling: Expecting the Unexpected. *Computer Languages*, 32(8):69–87, 1994.

[18] C. Schaffert et al. An Introduction to Trellis-Owl. In *OOPSLA'86*, pages 9–16, 1986.

[19] L. Ferreira and C. Rubira. The Design and Implementation of a Dependable and Distributed Framework for Train Controlers. Submitted to "Software: Practice & Experience".

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison Wesley Publishing Company, 1995.

[21] A. Garcia, D. Beder, and C. Rubira. An Exception Handling Mechanism for Developing Dependable Object-Oriented Software Based on a Meta-Level Approach. In *10th International Symposium on Software Reliability Engineering - ISSRE'99*, pages 52–61. IEEE Press, November 1999.

[22] A. Garcia, D. Beder, and C. Rubira. An Exception Handling Software Architecture for Developing Robust Software. Submitted to 5th IEEE International Symposium on High Assurance Systems Engineering, March 2000.

[23] A. Garcia and C. Rubira. Um Mecanismo Orientado a Objetos para Tratamento de Excecções em Software Concorrente Tolerante a Falhas. In *VIII Symposium of Fault-Tolerant Computing*, pages 33–47, Campinas, Brazil, July 1999.

[24] A. Garcia and C. Rubira. An Exception Handling Software Architecture for Developing Robust Software. In *2nd Workshop on Exception Handling in Object-Oriented Systems - ECOOP'2000*, June 2000.

[25] A. Garcia, C. Rubira, A. Romanovsky, and J. Xu. A Comparative Study of Exception Handling Proposals for Dependable Object-Oriented Software. Submitted to Journal of Systems and Software, March 2000.

[26] N.H. Gehani. Exceptional C or C with Exceptions. *Software: Practice and Experience*, 22(10):827–848, October 1992.

[27] C. Ghezzi and M. Jazayeri. *Programming Languages Concepts.* John Wiley & Sons, 3rd edition, 1997.

[28] A. Goldberg and D. Robson. *Smalltalk-80, the Language and its Implementation.* Addison-Wesley, Massachussetts, 1983.

[29] J.B. Goodenough. Exception Handling: Issues and a Proposed Notation. *Communications of the ACM*, 18(12):683–696, December 1975.

[30] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification - Version 1.0.* Addison-Wesley, 1996.

[31] M. Hof, H. Mossenbock, and P. Pirkelbauer. Zero-Overhead Exception Handling Using Meta-Programming. *Lectures Notes in Computer Science*, 1338:423–431, 1997.

[32] V. Issarny. Programming Notations for Expressing Error Recovery in a Distributed Object-Oriented Language. Technical Report 1822, IRISA/INRIA, Rennes, France, 1992.

[33] V. Issarny. An Exception-Handling Mechanism for Parallel-Object-Oriented Programming: Toward Reusable, Robust Distributed Software. *Journal of Object-Oriented Programming*, 6(6):29–40, October 1993.

[34] R.E. Johnson. Frameworks = Components + Patterns. *Communications of the ACM – Object-Oriented Application Frameworks*, 40(10):39–42, October 1997.

[35] A. Koening and B. Stroustrup. Exception Handling for C++. *Journal of Object-Oriented Programming*, 3(2):16–33, July/August 1990.

[36] S. Lacourte. Exceptions in Guide, an Object-Oriented Language for Distributed Applications. *Lectures Notes in Computer Science*, 512:268–287, July 1991.

[37] J. Lang and D. Stewart. A Study of the Applicability of Existing Exception-Handling Techniques to Component-Based Real-Time Software Technology. *ACM Computing Surveys*, 20(2):274–301, March 1998.

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

[39] H. Lieberman. *Concurrent Object-Oriented Programming in Act1*, pages 9–36. MIT Press, 1987.

[40] B.H. Liskov and A. Snyder. Exception Handling in CLU. *IEEE Transaction on Software Engineering*, SE-5(6):546–558, November 1979.

[41] M.D. Maclaren. Exception Handling in PL/I. *SIGPLAN Notices*, 12(3):101–104, March 1977.

[42] O.L. Madsen, B. Moller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*, chapter 16. Addison-Wesley Publishing Company, 1995.

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

[44] B. Meyer. *Object-Oriented Software Construction.* New York: Prentice-Hall, 1988.

[45] B. Meyer. *Eiffel - The Language.* Prentice Hall, 1992.

[46] R. Miller and A. Tripathi. Issues with Exception Handling in Object-Oriented Systems. In *Lectures Notes in Computer Science - ECOOP'97*, volume 1241, pages 85–103. Springer-Verlag, 1997.

[47] J. Mitchell, W. Maybury, and R. Sweet. *Mesa Language Manual.* Xerox Research Centre, 1979.

[48] S. Mitchell, A. Burns, and A. Wellings. MOPping up Exceptions. In *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, pages 365–366, 1998.

[49] D. Moreto. Monitoramento de Eventos Compostos em Sistemas Distribuídos. Master's thesis, IME - University of Sao Paulo, Brazil, September 1998. In Portuguese.

[50] G. Nelson. *Systems Programming with Modula-3.* Prentice Hall, 1991.

[51] A. Oliva and L. Buzato. Composition of Meta-Objects in Guaraná. In *Workshop on Reflective Programming in C++ and Java, OOPSLA'98*, pages 86–90, Vancouver, BC, Canada, October 1998.

[52] D. Papurt. The Use of Exceptions. *JOOP*, pages 13–17,32, May 1998.

[53] B. Randell, A. Romanovsky, R.J. Stroud, J. Xu, A.F. Zorzo, D. Schwier, and F. von Henke. Coordinated Atomic Actions: Formal Model, Case Study and System Implementation. Technical Report 628, Department of Computing Science, University of Newcastle upon Tyne, UK, 1998.

[54] B. Randell, J. Xu, and A. Zorzo. Software Fault Tolerance in Object-Oriented Systems Approaches. Technical Report 597, Department of Computing Science, University of Newcastle upon Tyne, UK, 1997.

[55] B. Randell and A. Zorzo. Exception Handling in Multiparty Interactions. In *VIII Symposium of Fault-Tolerant Computing*, Campinas, Brazil, July 1999.

[56] K. Renzel. Error Detection. In *EuroPLOP'97*, 1997.

[57] A. Romanovsky. Practical Exception Handling and Resolution in Concurrent Programs. Technical Report 545, Department of Computing Science, University of Newcastle upon Tyne, UK, 1996.

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

[59] A. Romanovsky. Extending Conventional Languages by Distributed/Concurrent Exception Resolution. *Journal of Systems Architecture*, pages 79–95, January 2000.

[60] A. Romanovsky, B. Randell, R.J. Stroud, J. Xu, and A. Zorzo. Implementing Synchronous Coordinated Atomic Actions Based on Forward Error Recovery. Technical Report 561, Department of Computing Science, University of Newcastle upon Tyne, UK, 1997.

[61] A. Romanovsky, J. Xu, and B. Randell. Exception Handling and Resolution in Distributed Object-Oriented Systems. In *Proc. the 16th Int. Conference on Distributed Computing Systems*, pages 545–553, Hong Kong, 1996.

[62] A. Romanovsky, J. Xu, and B. Randell. Exception Handling in Object-Oriented Real-Time Distributed Systems. In *International Symposium on Object-oriented Real-time Distributed Computing*, Kyoto, Japan, April 1998.

[63] C. Rubira. *Structuring Fault-Tolerant Object-Oriented Systems Using Inheritance and Delegation.* PhD thesis, University of Computing Science, October 1994.

[64] J. Schwille. Use and Abuse of Exceptions – 12 Guidelines for Proper Exception Handling. *Lectures Notes in Computer Science – Ada-Europe'93*, 688:142–152, 1993.

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

[66] S.T. Taft and R.A. Duff. Ada 95 Reference Manual: Language and Standard Libraries, international standard iso/iec 8652:1995(e). In *Lectures Notes in Computer Science*, volume 1246. Springer-Verlag, 1997.

[67] P. Thomas and R. Weedon. *Object-Oriented Programming in Eiffel*. Addison-Wesley, 1995.

[68] K. Vo, Y. Wang, P. Chung, and Y. Huang. Xept: A Software Instrumentation Method for Exception Handling. In *8th International Symposium on Software Reliability Engineering - ISSRE'97*, pages 60–69, Albuquerque, NM, USA, November 1997. IEEE Press.

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

[70] J. Xu, B. Randell, A. Romanovsky, R. Stroud, A. Zorzo, A. Burns, S. Mitchell, and A. Wellings. Cooperative and Competitive Concurrency in Fault-Tolerant Distributed Systems. In *DeVa 1st Year Report*, 1997.

[71] J. Xu, B. Randell, C. Rubira-Calsavara, and R.J. Stroud. Software Tolerance: Towards an Object-Oriented Approach. Technical Report 498, University of Newcastle upon Tyne, Novembro 1994.

[72] J. Xu, A. Romanovsky, and B. Randell. Exception Handling in Distributed Object Systems: from Model to System Implementation. Technical Report 612, Department of Computing Science, University of Newcastle upon Tyne, UK, 1997.

[73] S. Yemini and D.M. Berry. A Modular Verifiable Exception Handling Mechanism. *ACM Transactions on Programming Languages and Systems*, 7(2):214–243, April 1985.

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