UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MOTF:

META-OBJETOS PARA TOLERÂNCIA A FALHAS

por

Maria Lúcia Blanck Lisbôa

Tese submetida como requisito parcial para a obtenção do grau de
Doutor em Ciência da Computação

Prof. Dr. Paulo Alberto de Azeredo
Orientador

Porto Alegre, dezembro de 1995.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Lisbôa, Maria Lúcia Blanck

MOTF: Meta-Objetos para Tolerância a Falhas/ Maria Lúcia Blanck Lisbôa. - Porto Alegre: CPGCC da UFRGS, 1995.

171 p: il.

Tese (doutorado) - Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, 1995. Azeredo, Paulo Alberto de, orient.

1. Tolerância a Falhas em Software. 2. Orientação a Objetos. 3. Reflexão Computacional. I. Azeredo, Paulo Alberto de. II. Título.

Universidade Federal do Rio Grande do Sul

Reitor: Prof. Hélgio Trindade
Pró-Reitor de Pesquisa e Pós-graduação: Prof. Cláudio Scherer
Diretor do Instituto de Informática: Prof. Roberto Tom Price
Coordenador do CPGCC: Prof. José Palazzo Moreira de Oliveira
Bibliotecária-Chefe do Instituto de Informática: Zita Prates de Oliveira

Às minhas afilhadas,

Clarissa,
Fernanda,
Nicole.

Prefácio

A idéia primeira [..] foi para mim, a princípio, como que um sonho, como um desses projetos impossíveis, que se acariciam e se deixam voar; uma quimera que sorri, que exibe seu semblante feminino e logo em seguida distende as asas, subindo para um céu fantástico. Mas a quimera, como tantas quimeras, transforma-se em realidade; tem suas imposições e suas tiranias, às quais se é forçado a ceder.

Honoré de Balzac
A Comédia Humana,
Prefácio.*

A organização

Este texto é um registro abrangente de um projeto de pesquisa que partiu da premissa de que técnicas e mecanismos tradicionais de *tolerância a falhas em software* poderiam ter sua implementação simplificada pela abordagem *orientada a objetos*. Vários indícios reforçavam a viabilidade desta premissa, entre os quais a escassez de literatura sobre o assunto e particularidades do modelo de programação orientada a objetos. Uma das principais metas deste trabalho é promover a reutilização de componentes. No nível de abstração mais alto, encontra-se a reutilização de um domínio, seja o próprio domínio da aplicação ou o domínio de tolerância a falhas. Com vistas à separação de domínios, um terceiro componente foi incorporado a este trabalho: a reflexão computacional. Resumindo, a pesquisa ficou assim estruturada:

Domínio: **Tolerância a Falhas**

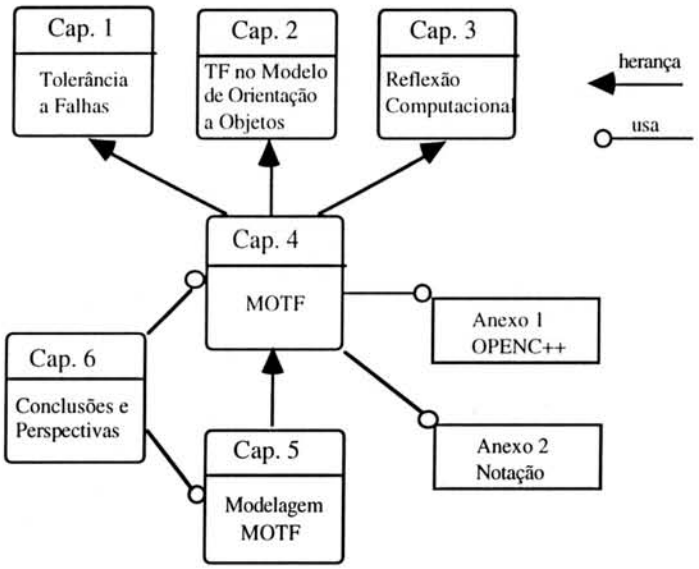
Modelo: **Orientação a Objetos**

Arquitetura: **Reflexão Computacional**

Em consonância com o modelo de orientação a objetos, o texto deste trabalho é estruturado em classes. As classes básicas são: a classe do domínio, a

* Ed. Globo,Vol XVII- Estudos filosóficos/ Estudos Analíticos, pág. 665

classe do modelo e a classe da arquitetura. Estes três componentes são focalizados separadamente nos capítulos 1, 2 e 3. A partir dessas classes, o framework Motf, por herança múltipla, especializa os comportamentos desejados. Assim, o capítulo 4 reune os três componentes, na abordagem dada neste trabalho: a proposta do framework MOTF - MetaObjetos para Tolerância a Falhas e o capítulo 5 identifica as classes e objetos que compõem o domínio de tolerância a falhas. O capítulo 6 dedica-se ao registro de conclusões. Esses componentes são organizados de forma hierárquica e se interrelacionam ao longo do texto, como esquematizado a seguir.



Complementam o texto o Anexo 1, dedicado à descrição da ferramenta de prototipação e o Anexo 2, que descreve a notação utilizada na modelagem.

A realização

A idéia primeira nasceu da convivência com minhas colegas Taisy da Silva Weber e Ingrid Eleonora Jansch-Porto. Ambas transmitiram para mim o seu entusiasmo pela área de Tolerância a Falhas, sua convição da existência de caminhos inexplorados nos aspectos de software e seu incentivo se tornou uma constante ao longo do desenvolvimento deste trabalho.

A idéia diretriz começou a tomar forma ao se direcionar para a área de Linguagens de Programação, à qual pertence meu paciente colega e orientador, Paulo Alberto de Azeredo.

A realização da idéia certamente teve as suas imposições, mas que se traduziram em benefícios. A imposição de interação com grupos de pesquisa externos foi duplamente benéfica: no aspecto pessoal, propiciou a convivência, o compartilhamento de idéias e experiências, pessoalmente e pela Internet com pesquisadores de outras universidades e nacionalidades, e no aspecto técnico, contribuiu significativamente para o refinamento do assunto. Meu estágio na Universidade de Gênova - Itália estabeleceu uma parceria proveitosa. Gabriella Dodero, Massimo Ancona, Vittoria Gianuzzi e Andrea Clematis contribuiram com idéias, dúvidas e se tornaram parceiros na escrita de artigos.

Parcerias locais também se estabeleceram. Com Alexandre Carissimi houve intercâmbio de idéias sobre implementação, buscando descobrir maneiras de concretizar, por meio de construções disponíveis em uma linguagem de programação, abstrações que se assemelhavam a truques de mágica. Gerson Geraldo Homrich Cavalheiro, recém graduado Mestre e a caminho de seu programa de doutoramento, se encarregou de instalar, testar, fazer protótipos e discutir idéias sobre implementação de reflexão computacional.

Muitas vezes a realização deste trabalho pareceu excessivamente distante, mas a confiança, a esperança, a expectativa e o incentivo de colegas, amigos e familiares que compartilham o meu quotidiano não permitiram que se tornasse inatingível. À minha lembrança imediata acorrem os nomes das amigas e colegas Nina Edelweiss, Lia Goldstein Golendziner, de meu irmão Dênis, meu sobrinho Luiz Eduardo e com especial carinho, meus pais, Herberto e Abigail.

Mas sobretudo, tenho sempre presente, ao longo de mais de duas décadas de vida acadêmica, a valiosíssima companhia de meu colega de trabalho, colega de curso de mestrado, meu amigo, meu marido Carlos Arthur Lang Lisbôa.

Muito obrigada a todos.

Sumário

Lista de figuras

Lista de tabelas

Resumo

As técnicas de programação e os mecanismos de linguagens de programação destinados ao desenvolvimento de aplicações de alta confiabilidade são agrupadas sob a denominação de *tolerância a falhas em software*. A área de tolerância a falhas abrange uma série de técnicas com funcionalidades e aplicabilidade bem definidas, permitindo que seja considerada um domínio próprio - o domínio de tolerância a falhas. O conteúdo de informação desse domínio não é auto-suficiente, uma vez que atua sobre outros domínios. Seu objetivo é garantir as funcionalidades das aplicações desenvolvidas em outros domínios.

Ao conjugar o domínio de tolerância a falhas a um outro domínio, ou seja, ao domínio de uma aplicação, o primeiro passa a se responsável pelos requisitos *não-funcionais* da aplicação. Os requisitos não funcionais de uma aplicação, a exemplo de confiabilidade e segurança, são cruciais em muitas aplicações e exigem métodos e conhecimentos que são distintos do domínio da aplicação.

O modelo de orientação a objetos incentiva o desenvolvimento de aplicações através da composição de objetos, cada qual com a sua estrutura e comportamento específicos. Cada particular composição de objetos forma um conjunto que deve observar um comportamento que atenda aos requisitos da aplicação, de forma confiável. Com o objetivo de aumentar a confiabilidade da aplicação e de minimizar o efeito de possíveis falhas do sistema, são propostos *objetos tolerantes a falhas*. Objetos tolerantes a falhas são objetos responsáveis por serviços críticos da aplicação e que possuem mecanismos que garantem a confiabilidade e disponibilidade destes serviços. Comportamentos tolerantes a falhas de objetos são obtidos por redundância de componentes, incluindo replicação e diversidade. O gerenciamento da redundância é executado de forma independente do domínio da aplicação e exercido em um meta-nível, através de técnicas de reflexão computacional.

A adoção de reflexão computacional no modelo de orientação a objetos permite organizar as atividades de tolerância a falhas sem interferir no aspecto estrutural dos objetos do domínio da aplicação. Os controles que devem ser exercidos pelos *meta-objetos* sobre os objetos da aplicação são

realizados em um meta-nível, de forma a separar as funcionalidades específicas da aplicação daquelas pertinentes ao domínio de tolerância a falhas. Estes *meta-objetos*, são organizados na forma de um *framework*, denominado MOTF - Meta-objetos para Tolerância a Falhas.

O projeto de MOTF é um *framework* que apoia o desenvolvimento de aplicações tolerantes a falhas, compreendendo múltiplas classes que definem as funcionalidades exigidas por diversas técnicas de tolerância a falhas. Adota uma arquitetura reflexiva, na qual o meta-nível é dedicado às atividades de detecção e recuperação de erros através da monitoração de objetos da aplicação, localizados no nível base. Características de tolerância a falhas podem ser adicionadas a objetos considerados críticos pela aplicação, assim distribuindo, e não centralizando, a propriedade de tolerar falhas entre objetos da aplicação. Incorporando os princípios de reflexão computacional ao modelo de orientação a objetos dois benefícios principais se salientam: promover a reutilização de objetos tolerantes a falhas e garantir a invulnerabilidade do objeto do domínio da aplicação, ao separar as ações pertinentes ao domínio da aplicação das específicas do sistema tolerante a falhas.

Palavras-Chave: Tolerância a Falhas em Software, Orientação a Objetos, Reflexão Computacional.

Title: "MOFT- METAOBJECTS FOR FAULT-TOLERANCE"

Abstract

Software fault-tolerance encompasses all techniques and programming languages' mechanisms intended to support the development of high reliability software. We can consider the fault-tolerance area a proper domain of knowledge composed by well-defined techniques used to guarantee the reliability of applications related to other domains. Therefore, the fault-tolerance domain acts over other domains.

When the fault-tolerance domain is merged into an application domain it becomes responsible for the non-functional requirements of the application. Among those requirements, reliability and safety are crucial ones and they use methods and concerns not related to the application domain.

The object-oriented approach to software development allows a software to be decomposed into a set of components - the objects. Each object has its own structure and behavior. The view of a system as composed by interacting objects can be quite convenient in expressing different degrees of fault tolerance. We can distinguish between critical and non-critical objects and we may even distinguish between critical and non-critical operations within a single object. The objective of this research is the exploitation of object-oriented approach to increase reliability and decrease the effects of failures based on the provision of fault-tolerant objects. Fault-tolerant objects are abstractions of high reliability components and are designed to support several fault-tolerance strategies.

Furthermore, computational reflection is adopted to organize fault-tolerant activities at a meta-level and to provide transparent interfacing among fault-tolerant and non-fault-tolerant objects. A fault-tolerant object can be defined as an object that represents a single interface to redundant services and whose behavior is controlled by a metaobject. Possible behaviors of fault-tolerant objects include replication or diversity and the associated metaobject adds a specific fault-tolerant behavior to its referent object without interfering in its internal structure.

MOTF - Metaobjects for Fault Tolerance is a framework intended to support the development of fault-tolerant applications. This framework consists of reusable meta-level classes. Each meta-level class implements a fault-tolerant service, and metaobjects are used as monitoring agents of fault-tolerant objects. The reflective object-oriented architecture promotes reusability and hides the programming of fault-tolerant mechanisms from the application.

Keywords: Software Fault-Tolerance, Object-Oriented, Computational Reflection.

Capítulo 1

TOLERÂNCIA A FALHAS

Discute aspectos de desenvolvimento de software tolerante a falhas, destacando aqueles de maior interesse no âmbito deste trabalho. Estabelece as premissas e delimita as pesquisas. Finaliza descrevendo trabalhos correlatos.

1.1 Introdução

Grande parte do processo de desenvolvimento de software é orientada pela especificação de requisitos. A especificação de requisitos explicita as funcionalidades do software e outras exigências tais como custo, performance, facilidade de uso, confiabilidade e outras mais, também conhecidas como requisitos não funcionais. Estes últimos constituem o foco de interesse deste trabalho, mais precisamente os que caracterizam um sistema digno de confiança[1].

Um sistema digno de confiança é caracterizado [HEI92], [LAP92] por: ser *disponível* (pronto para ser usado quando necessário), *confiável* (fornece continuamente um serviço adequado, *seguro* [2] (incapaz de causar danos ao seu ambiente) e *protegido*[3] (no sentido de preservar a sua confidencialidade).

Dentre as características acima, a confiabilidade é a mais freqüentemente associada ao domínio de tolerância a falhas. Produzir software capaz de se manter continuamente em operação e atendendo plenamente os seus requisitos é um objetivo que exige uma especial dedicação durante todo o processo de desenvolvimento. A demanda por software tolerante a falhas deixou de ser exclusiva do assim chamado software crítico, passando a ser uma qualidade desejável nas mais diversas aplicações, mesmo aquelas nas quais o risco de um defeito de software não cause grandes danos à vida humana, ao meio-ambiente ou à propriedade.

A busca da confiabilidade de software emprega diversos mecanismos, que objetivam garantir não apenas a sobrevivência do software, ou seja, que ele tenha a sua execução anormalmente interrompida (com ou sem mensagem indicativa), mas, acima de tudo, a obtenção do serviço especificado. De forma mais objetiva, a confiabilidade de um software é mensurável, conforme a definição de Anderson e Lee [AND81]:

> Def. 1.1: A confiabilidade de um sistema pode ser caracterizada por uma função R(t) que expressa a probabilidade que o sistema atenderá à sua especificação durante o período de tempo t.

---

[1] Do inglês *dependable system*.
[2] Do inglês *safe*.
[3] Do inglês *secure*.

Esta definição de confiabilidade ignora as conseqüências dos defeitos apresentados pelo software, sendo todo e qualquer defeito considerado de igual peso. A confiabilidade de um software não implica na sua total correção, mas apenas que ele forneça respostas aceitáveis, mesmo em presença de dados incorretos ou de ambientes adversos. Uma resposta é aceitável se os eventuais defeitos apresentados pelo software não perturbam sériamente a sua funcionalidade. Myers [MYE76] une os conceitos subjetivos e objetivos em uma única definição, quando se refere especificamente à confiabilidade de software:

> Def. 1.2: Confiabilidade de software é a probabilidade de que um determinado software executará por um particular período de tempo sem apresentar defeito, ponderada pelo custo ao usuário de cada defeito manifestado.

Desta definição é possível deduzir que, embora probabilisticamente mensurável, a confiabilidade de software tem exigências e gradações distintas para diferentes aplicações. A confiabilidade pode ser baixa, se a conseqüência de um defeito for apenas um inconveniente para o usuário (por exemplo, de um processador de texto); cresce para uma exigência média, se representar um risco de baixo custo social ou econômico (é o caso de inoperância temporária de um terminal bancário de consulta) e deve ser alta, se o mau funcionamento do software provocar catástrofes econômicas, ambientais, sociais ou riscos de vida (por exemplo, aplicações aviônicas). Este último ponto de vista é resumido por Sheldon [SHE92]:

> Def. 1.3: Confiabilidade é a probabilidade de operação livre de falhas por um determinado tempo, em um determinado ambiente e para um determinado objetivo.

A medida de confiabilidade é usada no processo de decisão para a liberação do software como um produto acabado, pronto para ser usado, estabelecendo a separação entre as atividades de eliminação de falhas e de tolerância a falhas. Naturalmente, nesta decisão deve ser ponderada a finalidade do produto e as possíveis conseqüências de um defeito operacional do software, bem como a relação custo-benefício de buscar uma maior confiabilidade.

É importante notar que a confiabilidade relaciona-se diretamente com a especificação de requisitos funcionais do sistema. A partir da premissa que os requisitos são corretos, as funcionalidades especificadas devem ser

atendidas. Com base na correção dos requisitos, são conduzidas as atividades de prevenção e eliminação de falhas e projetadas as técnicas de tolerância a falhas porventura utilizadas.

1.2 Tolerância a falhas em software

Os conceitos básicos, terminologia e definições usados nesta seção provém de Anderson [AND81], Jalote[JAL94] e Laprie[LAP92].

Tolerância a falhas compreende todas as técnicas e mecanismos empregados em sistemas de computação (hardware e software) com o objetivo de garantir a continuidade do fornecimento de seus serviços, em consonância com as suas especificações. Um sistema que continua a fornecer os serviços especificados, apesar da existência de falhas em seus próprios componentes ou em seu meio-ambiente, é dito tolerante a falhas. Sistemas tolerantes a falhas são sistemas que adotam redundância no espaço, por replicação de componentes de hardware ou software, e ou no tempo, por repetição de computações.

O termo *tolerância a falhas em software* abrange todas as técnicas de tolerância a falhas que podem ser implementadas em software, incluindo as destinadas a tolerar falhas de componentes de hardware [JAL94]. Um software capaz de mascarar falhas de seus próprios componentes é denominado de software tolerante a falhas.

Em software, o termo *falha*[4] é utilizado para caracterizar imperfeições introduzidas durante o seu processo de desenvolvimento, geralmente devido a algum engano, má-interpretação ou omissão relativa à especificação do software. É de senso comum que falhas existem em qualquer software. Ao longo do processo de desenvolvimento do software, em algum momento, algum membro da equipe de desenvolvimento ou alguma ferramenta de desenvolvimento (por falha própria ou mau uso), certamente irá gerar uma falha no produto, como abaixo definida.

> Def. 1.4: Falha de software é qualquer imperfeição presente no código executável de um programa, cuja manifestação possa implicar em diminuição de confiabilidade.

[4]Do inglês *fault*.

A falha é de natureza estática e estará presente em todas as cópias do software, deixando de existir apenas quando for eliminada. A eliminação de uma falha dá-se por simples remoção (por exemplo, de um ponto-e-vírgula que se reflete na estrutura de um bloco), por substituição (por exemplo, de um operador relacional que altera o resultado de uma condição) ou por alteração do projeto do sistema ou do projeto do programa [LIS94].

Mesmo presente, uma falha pode passar despercebida durante todo o processo de desenvolvimento do software e mesmo por longos períodos de sua vida útil pode permanecer latente, caso não ocorram as condições propícias à sua manifestação.

Entre as condições propícias à manifestação de uma falha, o meio-ambiente desempenha um papel importante e, portanto, deve ter a sua atuação bem delimitada. Um meio-ambiente desfavorável pode ocasionar diferenças de comportamento em duas cópias idênticas do mesmo software. Por exemplo, pequenas diferenças nas máquinas virtuais onde os softwares são executados podem ser suficientes para a manifestação de uma falha. Em relação à especificação do software, significa que ela deve possuir o atributo de completude, abrangendo todas as possíveis interações do software com seu meio-ambiente operacional e que atributos como portabilidade e proteção devem ser explicitados.

Ao manifestar-se, uma falha conduz a computação a um estado errôneo; ou seja, na origem de um erro de software sempre existe uma falha que abandonou seu estado de latência.

> Def. 1.5: Erro de software é uma manifestação de uma falha existente no software.

Na definição 1.5 destaca-se a possibilidade de detecção de erros, extremamente importante para o exercício de qualquer atividade de tolerância a falhas. Ao ocorrer um estado errôneo, este pode temporariamente impedir o prosseguimento da computação ou pode produzir resultados não condizentes com as especificações do software. Esses estados sempre podem ser detectados; a questão que se coloca é o custo associado.

Quando um estado errôneo não é detectado ou nenhuma ação é realizada para desfazer ou mascarar a ocorrência de erro em um

componente, o serviço desse componente deixa de atender a sua especificação.

Quando ocorre esta situação em que o comportamento visível do software se desvia do comportamento especificado, seja por omissão de resultados, antecipação ou retardo de respostas ou ainda por apresentar resultados incorretos, diz-se que ocorreu um *defeito*[5] de software e nenhuma atividade de tolerância a falhas pode mais ser executada, por ser tarde demais.

Neste ponto, é importante fazer uma distinção clara entre as tarefas de reparação de falhas, recuperação de erros e tolerância falhas.

- *Reparar uma falha* é uma atividade estática, onde se fazem necessárias alterações no código do programa, com o intuito de eliminar a falha geradora do erro.

- *Recuperar um erro* é uma atividade dinâmica, conduzida durante o processo de execução, com o intuito de alterar o estado errôneo momentâneo em busca de um estado normal de execução que permita o prosseguimento da operação, mesmo de forma degradada.

- *Tolerar falhas* em software consiste em incorporar componentes adicionais ao programa que proporcionem alternativas de funções consideradas críticas e algoritmos excepcionais, na forma de mecanismos de tratamento de exceções, que permitam a continuidade do processo de execução do software, frente à situações anormais que se apresentem durante a sua operação.

As atividades de prevenção, eliminação e de tolerância a falhas devem ser uma preocupação constante durante todo o processo de construção do software [HEI92]. A prevenção e a eliminação de falhas são perseguidas através do emprego de boas técnicas de engenharia de software e a tolerância a falhas deve ser objeto de especificações próprias, ou seja, *especificações de tolerância a falhas*. A especificação de tolerância a falhas inclui, entre outros, a especificação de restrições de segurança, restrições de projeto funcional e restrições de código [LEV91]. As restrições de segurança ponderam os riscos e seus limites, as restrições de projeto identificam ítens de segurança crítica (processos, dados e estados) e as restrições de código asseguram a adequada proteção dos componentes críticos.

---

[5] Do inglês *failure*

1.3 Níveis de tolerância a falhas

As atividades de tolerância a falhas necessitam o apoio da linguagem de programação de implementação adotada no sistema de controle de aplicações tolerantes a falhas, ou da própria aplicação.

A importância da existência de mecanismos em linguagens de programação para apoiar e simplificar o processo de desenvolvimento foi recentemente realçado por Schlichting e Thomas [SCH95], na linguagem FT-SR[6].

A tolerância a falhas também deve encontrar respaldo no ambiente de suporte da aplicação e ou seu sistema de controle e no hardware de suporte, ou seja, em sua máquina virtual, em uma estrutura composta por diversas camadas.

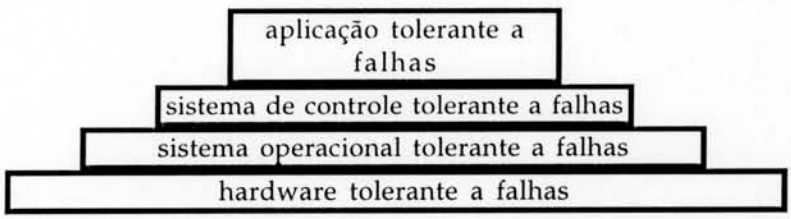


Figura 1.1 - Níveis de tolerância a falhas

A dinâmica de tolerância a falhas pode envolver todos os níveis mencionados na figura 1.1, cada nível contribuindo com a sua parcela para a consecução do objetivo comum - a contínua obtenção do serviço desejado em tempo hábil. A título de ilustração, a contribuição individual de cada nível poderia ser:

- o hardware mascara uma falha de memória transferindo os dados para outra área;
- o sistema operacional detecta um periférico defeituoso, isola-o e passa a utilizar um periférico alternativo;
- o sistema de controle faz inspeções sobre possíveis violações de tempo em um sistema distribuído e ativa um processo alternativo em outro processador;
- a aplicação detecta um operação ilegal, por meio de um mecanismo de tratamento de exceções e consegue anular o seu efeito, prosseguindo a execução.

---

6 Ver a seção Trabalhos Correlatos, neste capítulo.

A construção de aplicações tolerantes a falhas tem uma atuação limitada no tocante à aplicação, exigindo suporte de sua máquina virtual principalmente quanto concorrência, distribuição e restrições temporais são utilizadas.

Esta interdependência dos componentes se expressa no conceito de *dependability* [HEI92] empregado para sistemas dignos de confiança, dos quais seus usuários podem depender mesmo em situações adversas. Um componente de um sistema é dito dependente de outro quando a correção de sua execução requer o comportamento correto do segundo componente. A identificação de dependências entre os componentes de um sistema é uma das atividades precípuas do projeto e verificação de sistemas que exigem alta confiabilidade.

Kim [KIM95] classifica a tolerância a falhas de sistemas em cinco categorias, conforme ilustrado no quadro a seguir.

Tabela 1.1 - Graus de tolerância a falhas

Grau	Danos previstos	Capacidade de recuperação
<4>	Sem perda de ações visíveis	Tolerância a falhas a nível de ação (recuperação de uma ação visível interrompida)
<3>	Perda de uma ou mais ações visíveis	Lenta recuperação de um serviço (sem perda de hardware)
<2>	Perda de um ou mais serviços	Recuperação parcial de hardware (degradação de serviços)
<1>	Perda de todos, exceto um núcleo mínimo de serviços críticos	Recuperação mínima de um núcleo de hardware (serviços críticos mínimos)
<0>	Perda de serviços críticos	Nenhuma tolerância a falhas

Outro aspecto relevante a ser considerado na tomada de decisão sobre quais componentes devem ser implementados com mecanismos de tolerância a falhas é o da possibilidade de falha em um componente não crítico do sistema, mas cujo mau funcionamento pode comprometer todo o sistema. Lembrando o ditado 'nenhuma corrente é mais forte que o mais fraco de seus elos', um sistema somente pode ser digno de confiança se todos os seus componentes individuais forem confiáveis. Este aspecto tem a sua importância realçada na norma MOD00-55 [PLA93] que estabelece critérios para desenvolvimento de sofware de segurança crítica para sistemas do Ministério da Defesa do Reino Unido: o sistema deve ser

projetado de modo que o software de segurança crítica seja isolado de outros componentes. Isto é particularmente importante para o software que embora não implemente nenhuma função de segurança, mas caso apresente defeito e esteja fortemente acoplado a um software de segurança crítica, possa causar falhas neste último.

1.4 Aplicações tolerantes a falhas

Uma aplicação compreende um conjunto de componentes interrelacionados. Estes componentes são organizados na forma de funções, procedimentos, módulos, cláusulas ou objetos, de acordo com o modelo ou linguagem de implementação. Sua composição no texto pode ser em seqüência ou aninhada, e sua execução pode ser puramente seqüencial ou por requisição.

Independentemente de sua organização, composição ou forma de execução, um componente é individualmente caracterizado por sua funcionalidade, determinada por uma função de mapeamento entre seus dados iniciais e os resultantes de sua execução. É precisamente esta funcionalidade que deve ser assegurada por *componentes tolerantes a falhas*, também denominados de componentes *ideais* tolerantes a falhas por Lee e Anderson [LEE90].

Componentes são tolerantes a falhas, ou não, desde a sua gênese - a especificação de requisitos. Uma vez identificados os componentes cuja criticalidade[7] dos serviços fornecidos exigem alta confiabilidade, devem ser especificadas as propriedades de tolerância a falhas desejáveis em cada componente. As propriedades dependem da criticalidade do serviço e de suas características individuais; componentes de uma mesma aplicação podem exigir o emprego de técnicas distintas.

Assegurada a tolerância a falhas a nível de componentes, deve ser avaliada a necessidade de tornar tolerantes a falhas grupos de componentes que, por sua sua composição e formas de interação, contribuem para a consecução de um serviço crítico. Oportunamente, Birmann [BIR93] coloca:

> A expectativa de que a confiabilidade de um sistema distribuído corresponda diretamente à confiabilidade de seus constituintes nem sempre é verdadeira. Os mecanismos usados para estruturar um

7 Refere-se a componentes decisivos, de cujo comportamento o sistema é dependente.

sistema distribuído e para implementar a cooperação entre os componentes desempenham um papel vital na determinação da confiabilidade de um sistema.

Um grupo de componentes pode tolerar falhas de sistemas (ex. replicado), ou preservar a integridade e consistência de seu serviço (ex. transação atômica) ou garantir longa vida a seus dados (ex. persistência).

A seguir são abordados componentes com propriedades de tolerância a falhas em ambientes centralizados e distribuídos.

1.4.1 Cenário 1: Ambientes centralizados e não concorrentes

Na execução puramente seqüencial, os componentes são executados na ordem em que se encontram no texto enquanto que na execução por requisição um componente requisita a execução de outro componente, interrompendo a sua execução enquanto aguarda o término da execução do componente requisitado. A figura 1.2 esquematiza o fluxo de execução por requisição e serve como cenário básico para introduzir a idéia de componentes com propriedades de tolerância a falhas.

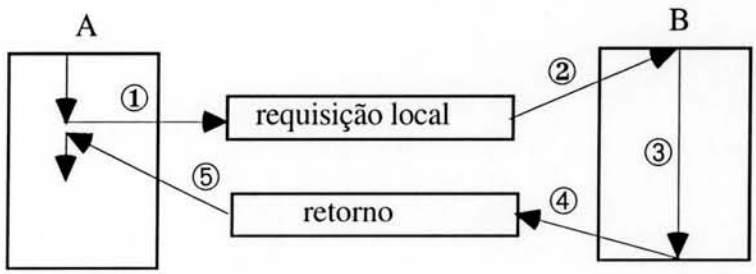


Figura 1.2 - Cenário de execução

Neste cenário destacam-se os seguintes pontos demarcados na figura e que indicam momentos de execução.

① O componente A requisita a execução do componente B, por meio de uma requisição (i.e., chamada de procedimento, mensagem) parametrizada.

② O componente B recebe a requisição e inicia a sua execução.

③ O componente B executa as sua operações

④ O componente B termina a sua execução e retorna resultados.

⑤ O componente A recebe os resultados da execução de B e retoma a sua execução no ponto de interrupção.

Em cada um desses momentos pode-se imaginar a manifestação de uma falha, como por exemplo:

Momento ①: Falha nos parâmetros de requisição. Ex. Quantidade incorreta de parâmetros.

Momento ②: Falha nos parâmetros de requisição. Ex. Tipo incorreto de parâmetro.

Momento ③: Falha em operação interna de B. Ex. divisão por zero.

Momento ④: Execução com erro. Ex. Retorna indicação de erro.

Momento ⑤: Resultados incorretos da execução de B. Ex. Valor fora dos limites.

A correlação causa-efeito de erros de software foi abordada por Nakajo e Kume [NAK91], mostrando que:

- Falhas de interface e falhas funcionais de módulos respondem juntas por cerca de 90% do total de falhas identificadas em quatro produtos comerciais.

- Falhas funcionais em módulos são estritamente relacionadas com a especificação funcional do módulo, isto é, projetar software que satisfaça as necessidades do usuário e, destas, 45% devem-se à má-interpretação dos requisitos.

Algumas destas falhas podem não se manifestar de forma evidente, mas em sua decorrência uma outra falha pode se manifestar. Por exemplo, em presença de funções polimórficas com amarração por parâmetros, a quantidade incorreta de parâmetros pode resultar na execução de outro componente homônimo e, em conseqüência, no fornecimento de resultados incorretos. Mas certamente, algumas falhas podem ser antecipadas e providências podem ser tomadas no sentido de detectá-las e, na medida do possível, mascará-las. Neste caso destacam-se:

Falhas em operações internas de componentes: a detecção e o tratamento de condições anormais pode ser abordado através de sistemas de tratamento de exceções. Sistemas de tratamento de exceções possuem interpretações distintas para ambientes modulares [AZE80], orientação a objetos [CAR92] e orientação a dados [CUI92]. São importantes mecanismos de

linguagens de programação para a construção de programas *robustos* e de aplicações tolerantes a falhas [CRIS82], [RUB4b].

Falhas de projeto de componentes: os desvios de comportamento de componentes em relação à sua especificação são abordados por técnicas de tolerância a falhas baseadas em redundância de componentes. A redundância temporal, baseada em computações repetidas, seja reexecutando o mesmo código ou código distinto de igual funcionalidade, permite a detecção e recuperação de erros de software.

A redundância temporal de componentes pode ser implementada de forma estática ou dinâmica [XU94]. A redundância estática usa componentes adicionais de software, denominados versões ou variantes, todas executadas concorrentemente de forma a mascarar os efeitos de erros de software.

O exemplo clássico de redundância estática é a programação em n-versões; sua descrição pode ser encontrada em [AVI85]. A redundância dinâmica consiste em diversos componentes redundantes entre os quais um subconjunto, normalmente unitário, está ativo num determinado momento. Quando um erro de software é detectado no componente ativo (por exemplo, através da sinalização de uma exceção ou resultado não aceito), ele é substituído por outro componente. Exemplos de técnicas de redundância dinâmica incluem blocos de recuperação [RAN75], programação auto-verificadora [LAP87] e re-expressão de dados [JAL94]. Esta última técnica não exige múltiplas versões de componentes, utilizando re-execução do mesmo componente falho. Baseia-se na premissa que computações falham devido aos dados de entrada e que uma alteração nestes dados de entrada pode resultar em uma computação bem sucedida do mesmo código.

O emprego de componentes redundantes, além de exigir a implementação dos componentes adicionais, implica na existência de um sistema de gerenciamento destes componentes, particularizado por técnica. Uma descrição mais detalhada das técnicas mencionadas e seu gerenciamento encontram-se em [LIS94].

1.4.2 Cenário 2: Ambientes distribuídos

Introduzindo no cenário anteriormente descrito a possibilidade de distribuição de componentes, a requisição local de execução deve ser substituída por uma requisição remota. Uma das formas mais simples de

requisição, a chamada de procedimento remoto[8] se constitui num mecanismo síncrono de interação: o componente requisitante permanece bloqueado enquanto aguarda o resultado do componente remoto requisitado. Este mecanismo é uma abstração semelhante à chamada de procedimentos locais, mas sua implementação é mais complexa, devido a dois fatores principais: transmissão de parâmetros sem acesso à memória compartilhada e possibilidade de falha no sistema no transcurso de uma chamada, ocasionando defeito de omissão, desempenho ou de resposta no resultado esperado no ponto de chamada.

Abstraindo do ambiente de suporte das aplicações distribuídas e considerando que a comunicação entre os componentes é confiável a ponto de permitir que qualquer mensagem enviada por um componente alcance o seu destino sem sofrer corrupção, o foco de atenção pode se concentrar apenas na aplicação. Neste caso, outras técnicas de tolerância a falhas podem ser utilizadas no âmbito da aplicação, destacando-se a redundância espacial por replicação de componentes, a preservação da integridade dos dados por meio de *ações atômicas* e do estado da computação, por meio de *conversações* .

A redundância espacial, que consiste em manter cópias fisicamente separadas de recursos, funções ou dados é usada principalmente para a detecção e mascaramento de falhas de recursos físicos, como processadores, memória e linhas de comunicação. Em [CRIS91] existe uma descrição dos tipos de falhas típicas de sistemas distribuídos.

1.4.3 Combinação de técnicas

As técnicas mencionadas podem ser usadas individualmente ou podem ser combinadas. A distribuição física dos equipamentos permite uma abordagem de tolerância a falhas que contempla, simultaneamente, a tolerância a falhas do sistema, por replicação simples de processos em processadores distintos (redundância espacial) e a tolerância a falhas de software, por programação diversitária (redundância temporal). Componentes podem ter seu *backup* em outro nodo da rede, ao mesmo tempo que alternativas de componentes de funcionalidade crítica podem coexistir em um nodo. A figura 1.3 ilustra esta metodologia, com dois

---

[8] RPC - *Remote Procedure Call.*

processos principais, P_1 e P_2, suas alternativas (a_1, a_2 e a_3) e (b_1, b_2 e b_3) e suas réplicas P_1' e P_2', respectivamente.

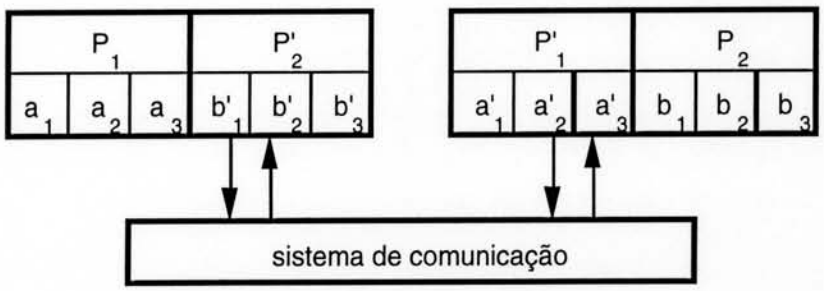


Figura 1.3 - Combinação de técnicas de tolerância a falhas.

A combinação de técnicas para a construção de software tolerante a falhas pode exigir a integração de diversos componentes do sistema [KIM95], [BIR93], [SHR94], [STR91].

Kim [KIM95] desenvolveu o esquema DRB - Distributed Recovery Blocks, uma das mais conhecidas técnicas que combina redundância espacial e temporal utilizando processos paralelos. Shrivastava [SHR94] utiliza ações atômicas aliadas a objetos persistentes para a construção de aplicações distribuídas[9]. Strigini [STR91] propõe a combinação de conversações com transações atômicas, objetivando a construção de componentes servidores tolerantes a falhas em seus dados e nas interações com outros componentes.

1.5 Objetivos e delimitação do trabalho

Este trabalho é centrado em sistemas de controle e gerenciamento de componentes redundantes no modelo de orientação a objetos, utilizando uma arquitetura reflexiva para apoiar a construção de aplicações tolerantes a falhas. Neste modelo um componente é organizado como um objeto que reúne dados e operações sobre esses dados e que interage com outros componentes (i.e., outros objetos) através de interfaces estáveis.

Componentes organizados na forma de objetos apresentam características peculiares que validam o esforço de desenvolvimento de uma pesquisa sobre o enfoque de tolerância a falhas particularizado para linguagens orientadas a objetos: a possibilidade de replicação por

9Ver a seção Trabalhos Correlatos, neste capítulo.

instanciação, a forte modularização com protocolos bem definidos, o intercâmbio de mensagens entre objetos, a herança comportamental, entre outros, tornam este modelo atraente para o desenvolvimento de programas tolerantes a falhas.

Não menos importante são as questões relacionadas com a originalidade e a importância do tema. No início das pesquisas foi constatada a escassez da literatura sobre o tema. Embora escassa, na literatura existente sobre tolerância a falhas no modelo de orientação a objetos podem ser encontrados trabalhos importantes nesta área, destacando-se o sistema operacional ISIS [BIR93] baseado em objetos que apoia o desenvolvimento do sistemas distribuídos tolerantes a falhas ao adotar protocolos confiáveis de distribuição de mensagens.

Recentemente foram divulgados os trabalhos de Rubira [RUB94a], Xu[XU94] e Fabre [FAB95], reforçando a importância do tema. A questão da originalidade é assim colocada por Xu [XU94]:

> Pelo que sabemos, poucas pesquisas tem explorado os benefícios potenciais e possíveis problemas de usar técnicas orientadas a objetos para facilitar o projeto de tolerância a falhas e para lidar com diversidade.

A premissa inicial deste trabalho é que algumas atividades de tolerância a falhas podem ser supridas por um nível superior ao da aplicação desenvolvida de acordo com o modelo de orientação a objetos, separando as ações de tolerância a falhas das ações do domínio da aplicação.

O principal objetivo dessa separação é facilitar o desenvolvimento de aplicações tolerantes a falhas, permitindo a adição de propriedades (não-funcionais) de tolerância a falhas sem alterar a funcionalidade do objeto a nível de aplicação. Não sendo possível a obtenção de total transparência na construção de tais aplicações, busca-se simplificar o processo de desenvolvimento. Classes de objetos fornecem os mecanismos principais para a implementação de atividades de tolerância a falhas, os quais podem ser selecionados, particularizados e utilizados pelos objetos que compõem a aplicação e, principalmente, reutilizados.

Na arquitetura reflexiva, meta-objetos realizam computações próprias do sistema sobre objetos de nível base. Neste trabalho é investigada a utilização de meta-objetos para prover diversos serviços pertinentes a atividades de tolerância a falhas e como forma de promover a transparência

e a reutilização. As técnicas e mecanismos utilizados são os encontrados na literatura sobre tolerância a falhas, adaptados ao modelo de orientação a objetos e às técnicas de reflexão computacional.

Outrossim, o modelo de orientação a objetos promove a utilização de ambientes de desenvolvimento de aplicações de domínios específicos, conhecidos como *frameworks* [TSI92]. Os *frameworks* amparam-se no conceito de reutilização, sob forma de classes abstratas que podem ser particularizadas para adequar-se às aplicações e seu ambiente de suporte, facilitando o desenvolvimento de aplicações por composição de objetos já existentes. Técnicas de tolerância a falhas tradicionalmente tem sido objeto de ambientes específicos para o desenvolvimento de aplicações tolerantes a falhas [PUR91], [AVI85].

Nesta abordagem, o projeto de MOTF- Meta-Objetos para Tolerância a Falhas é um *framework* composto por múltiplas classes de meta-objetos que suportam as funcionalidades exigidas pelos diversos métodos e técnicas do domínio de tolerância a falhas. O MOTF atua como agente de reflexão do domínio da aplicação para o domínio de tolerância a falhas, sendo este último considerado como um meta-nível do domínio da aplicação tolerante a falhas.

1.6 Trabalhos correlatos

Nesta seção são descritos brevemente trabalhos correlatos no âmbito de tolerância a falhas, classificados por assunto.

1.6.1 Tolerância a falhas no modelo de orientação a objetos

O trabalho de Rubira [RUB94a] consiste de um *framework* genérico para tolerância a falhas baseado classes abstratas que representam especificações de serviços de objetos e subclasses destas derivadas usadas para instanciação de variantes. O controle de redundância (i.e., blocos de recuperação e n-versões) é feito por funções genéricas parametrizadas.

Xu [XU94] apresenta um esquema conceitual para a recuperação de erros coordenando diversos objetos que interagem em ambientes concorrentes utilizando as técnicas de conversações e transações atômicas.

Recentemente Fabre [FAB95] também apontou as vantagens da arquitetura reflexiva para a implementação de técnicas de tolerância a falhas

em sistemas distribuídos, sugerindo a adoção de meta-objetos para monitorar objetos distribuídos, replicados de forma ativa, semi-ativa e passiva.

1.6.2 Ambientes de desenvolvimento

RMP Recovery Metaprogram, proposto por Ancona et al. [ANC90], implementa ações e mecanismos de tolerância a falhas através da monitoração do programa de aplicação. O programa de aplicação é visto como um conjunto de processos, dos quais um subconjunto é selecionado, pela própria aplicação, para incorporar mecanismos de tolerância a falhas. Estes mecanismos incluem blocos de recuperação, programação em n-versões e conversações. A aplicação indica através de diretivas a porção de código (início e fim) que deve ser controlado pelo RMP, as alternativas de execução e o teste de validação destas alternativas. De posse destas informações RMP especifica o fluxo de execução adicional para a implementação de pontos de recuperação, retrocesso do estado da computação e execução de alternativas.

O modelo de tolerância a falhas adotado pelo RMP é limitado às falhas de software originadas na aplicação e para as quais são previstas ações de recuperação por redundância. Falhas de hardware e no próprio núcleo devem ser mascaradas em seu próprio nível, assim como exceções e seu tratamento. As exceções não são sinalizadas ao programa de recuperação e também não são admitidas exceções dentro do RMP. Sua área de domínio são aplicações concorrentes, compostas por conjuntos de processos autônomos, independentes da linguagem de programação usada para escrever a aplicação. A arquitetura do RMP foi implementada através de um protótipo, com o único objetivo de validar a proposta, não estando portanto disponível para a utilização em desenvolvimento de aplicações.

POLYTH Este ambiente de desenvolvimento de software que suporta a programação tolerante a falhas é apresentado por Purtilo e Jalote [PUR91]. A unidade básica para a aplicação de técnicas de programação diversitária (blocos de recuperação ou n-versões) é um módulo (procedimento ou função), sendo os usuários livres para empregar estas técnicas em diferentes partes do sistema. O projetista especifica as interfaces via uma linguagem de especificação (MIL- module interconnection language) e outras propriedades de cada módulo, mostra

como eles são interconectados e, se tolerância a falhas é desejada então anota a estrutura com uma descrição da técnica a ser usada. O ambiente permite a distribuição do software de aplicação em arquiteturas heterogêneas, sendo a transformação de dados entre as diversas máquinas e diversas linguagens de programação feita de forma transparente à aplicação.

1.6.3 Ambientes distribuídos

ARJUNA É um ambiente de programação distribuída baseado em objetos com propriedades de atomicidade e persistência. Ações atômicas controlam seqüências de operações sobre objetos locais e remotos, que são instâncias de classes C++ [SHR95]. O modelo de computação adotado é conhecido como modelo de *objetos e ações*; programas de aplicação utilizam objetos persistentes (longa duração) sob o controle de ações atômicas (transações atômicas). Para a construção de aplicações distribuídas tolerantes a falhas, Arjuna fornece uma biblioteca de classes C++ , a partir da qual objetos da aplicação podem herdar propriedades.

ISIS É um sistema que fornece diversas ferramentas para a construção de aplicações distribuídas de alta confiabilidade. Orientado a processos, ISIS [BIR93] permite a construção de grupos dinâmicos de processos, oferecendo suporte para: composição de grupos, difusão confiável de mensagens, coleta de resultados (0, 1, Quorum, ALL), replicação de dados, redundância ativa de componentes, monitoração de processos e recuperação automática de grupos de processos.

RIO Reconfigurable Interconnectable Objects [BRI93], é um ambiente gráfico para a construção, configuração e operação de sistemas distribuídos. As técnicas de tolerância a falhas fornecidas pelo ambiente são baseadas em replicação: replicação ativa e replicação passiva. A especificação de tolerância a falhas é feita pelo usuário, através de uma linguagem de configuração. Esta linguagem parametrizada permite especificar o tipo de técnica pretendida, o número de réplicas desejadas e as estações preferenciais para a manutenção da réplicas.

O ambiente RIO é independente de linguagem de programação, exigindo apenas que os componentes do sistema sejam módulos, compostos por uma parte de código e uma parte de dados e um ou mais pontos de conexão externos. É baseado em objetos, associando a cada módulo os conceitos de classe e instância. Instância é relacionada com a unidade operacional e classe ao tipo que é utilizado para a criação desta unidade.

Classes podem ser compostas por outras classes, que podem ser instanciadas e interconectadas, configurando a aplicação distribuída.

1.6.4 Linguagens de programação

FT-SR — Extensão da linguagem SR com mecanismos para replicação de componentes, protocolos de recuperação e notificação de falhas de forma síncrona ou assíncrona [SCH95]. A implementação dos mecanismos de tolerância a falhas é feito em dois níveis: extensões no compilador e no ambiente de suporte. A máquina virtual de FT-SR reside em um núcleo separado, contendo primitivas para a criação, destruição e monitoração de componentes e de grupos de componentes, assim como serviços de tratamento e recuperação de erros.

FTCC — Fault-Tolerant Concurrent C [CME88] é uma extensão da linguagem C com mecanismos de programação paralela e replicação de processos. A tolerância a falhas é realizada através da criação de processos replicados, detecção de falhas em processos servidores seguida de comunicação ao processo cliente e terminação automática de processos *órfãos*.

1.6.5 Comentários sobre os trabalhos correlatos

a) Tolerância a falhas no modelo de orientação a objetos:

Estes trabalhos são baseados em características do modelo de orientação a objetos como herança, amarração dinâmica e funções genéricas, não adotando, apenas sugerindo, a arquitetura reflexiva.

O trabalho de Fabre [FAB95] é o que mais se aproxima da pesquisa aqui descrita, pois utiliza reflexão computacional no modelo de orientação a objetos para a implementação de protocolos de replicação ativa, semi-ativa e passiva em ambientes distribuídos.

b) Ambientes de desenvolvimento

A arquitetura de RMP- Recovery Metaprogram utiliza dois níveis de execução: o nível de aplicação e o nível de monitoração. Este modelo serviu de base para a arquitetura de MOTF- Meta-Objetos para Tolerância a Falhas [CLE95]. Este último diferencia-se por adotar o modelo de programação orientado a objetos e a arquitetura em níveis por reflexão computacional.

c) Ambientes distribuídos

Ambientes distribuídos com características de tolerância a falhas podem ser usados para suporte de execução distribuída dos *objetos tolerantes a falhas* aqui propostos. A princípio, o mais adequado é o sistema Arjuna, por ser orientado a objetos e escrito na linguagem C++, utilizada como linguagem de prototipação das classes dos meta-objetos descritas neste trabalho.

d) Linguagens de programação

Mostram a importância, sob o ponto de vista de linguagens de programação, da disponibilidade de mecanismos de tolerância a falhas. O *framework* MOTF não sugere nenhum novo mecanismo de extensão de linguagens de programação; baseia-se em características do modelo de orientação a objetos e utiliza a técnica de reflexão computacional.

Capítulo 2

TOLERÂNCIA A FALHAS NO MODELO DE ORIENTAÇÃO A OBJETOS

Salienta as características do modelo de orientação a objetos que motivam a sua interpretação particularizada para o domínio de tolerância a falhas. Inicia discorrendo sobre o modelo conceitual, prossegue justificando a sua adoção e se desenvolve analisando as suas características individuais. Conclui resumindo as contribuições do modelo de orientação a objetos ao desenvolvimento de aplicações tolerantes a falhas.

2.1 A revolução dos objetos

O sucesso do paradigma de orientação a objetos tem como principal apelo a reutilização. O velho conceito de macro-instrução como componente reutilizável foi lapidado à perfeição, unindo fortemente os conceitos de dados e ações sobre estes dados na forma de objetos da computação e comportamentos associados.

A equação de Wirth [WIR73] "programa = estruturas de dados + algoritmos" foi reinterpretada a nível de objeto, aumentando a abstração e encapsulando as dependências entre dados e ações. A abstração procedimental, nascida nas linguagens imperativas orientadas a procedimentos para modularizar a estrutura de programas de acordo com o princípio de "dividir para conquistar" e com o objetivo precípuo de reduzir a complexidade da computação, se tornou ainda mais abstrata com o mecanismo de encapsulamento.

Assim, a nova geração de projetistas e programadores de sistemas dispõe das facilidades deste fascinante paradigma, mas que também cobra as suas exigências. Este modelo exige uma reformulação das técnicas clássicas de projeto de sistemas e de programas, técnicas de programação, ambientes e ferramentas de desenvolvimento para que possa ser plenamente explorado em suas potencialidades.

O seu modelo conceitual também se espraia no ambiente de suporte: novos sistemas operacionais são projetados para suportar plenamente o conceito de objetos, a exemplo de Apertos [YOK92], Chorus [LEA93] e Choices [CAM93]; linguagens de programação são criadas de acordo com o modelo original, como a pioneira e sempre renovada SmallTalk [GOL83], ou são reinterpretadas para suportar as características do modelo, como C++ [STRO91] e ADA9X [DoD93]; e continuam surgindo interpretações sobre como objetos podem ser distribuídos, replicados e tornados persistentes [SHR94].

A agora vasta literatura sobre o modelo de orientação a objetos faz notar que não é trivial projetar uma aplicação e menos fácil ainda promover a reutilização de objetos [BOO94]. O alto grau de abstração e generalização exigido para a criação de classes de objetos reusáveis tem desafiado projetistas e programadores. Por outro lado, os repositórios de objetos reusáveis criados para facilitar o desenvolvimento de aplicações por

composição de objetos já existentes, às vezes se tornam super povoados, exigindo um paciente trabalho de 'navegação' em busca do objeto mais adequado à aplicação.

Uma solução para promover a reutilização tem apontado para a especialização por domínios, na forma de *frameworks*. Um *framework*, na interpretação de Tsichritz [TSI92], é uma coleção de classes de objetos junto com seus respectivos projetos e ferramentas que tornam possível a sua reutilização num determinado domínio de aplicação. A exigência dos projetos associados às classes de objetos promove a reutilização em mais alto grau, tanto ao possibilitar a especialização comportamental por herança, como ao permitir a reutilização do próprio projeto de classes abstratas cuja concretização depende de características particulares do ambiente de execução. Peter Wegner [WEG92] usa o termo *sistema fortemente aberto* para designar sistemas orientados a objetos, visto que o comportamento destes sistemas pode ser facilmente modificado de forma estática e ou dinâmica. O mesmo autor define as classes de objetos como componentes abertos/fechados por serem abertos aos clientes que desejam estendê-los e fechados aos clientes que querem usá-los.

2.2 O enfoque das linguagens de programação

Com uma população estimada em mais de mil exemplares [FRI92], as linguagens de programação são tradicionalmente classificadas através de dois métodos:

- Método evolutivo: distribui as linguagens em gerações, onde cada geração representa uma diferente tecnologia de software.
- Método de classes: as linguagens são separadas conforme o seu modelo computacional.

Nesta última classificação, são definidos prioritariamente dois modelos [PET90]:

- Modelo imperativo: as linguagens expressam seqüências de comandos que devem ser executados para a obtenção de um resultado.
- Modelo declarativo: as linguagens não possuem comandos, apenas "roteiros" que definem o quê deve ser computado, de forma independente das manipulações que devem ser feitas para a obtenção dos resultados.

A partir dos modelos primitivos - o imperativo e o declarativo - podem ser decompostos novos modelos, ou subclasses, que são conhecidos como paradigmas de linguagens de programação, entre os quais destacam-se:

- Subclasses do modelo imperativo

  Orientadas a procedimentos: as ações que se interrelacionam para a consecução de um objetivo são organizadas em módulos compostos por procedimentos hierarquicamente estruturados.

  Orientadas a objetos: as ações e seus dados são encapsulados em classes instanciáveis em objetos. A computação é vista como uma interação entre objetos, que se comunicam entre si através de mensagens.

- Subclasses do modelo declarativo

  Modelo funcional: as ações são organizadas como funções, que agem sobre dados locais, não provocando efeitos colaterais. A computação é baseada no conceito matemático de função, que é o mapeamento de um conjunto domínio em um conjunto contra-domínio.

  Modelo lógico: as ações são organizadas em cláusulas, que determinam objetivos a serem atingidos com base nos dados. A computação é baseada em lógica de primeira ordem, consistindo em um conjunto de axiomas que devem ser provados para alcançar um objetivo.

Embora seja comum considerar uma classe como equivalente a um módulo tradicional de linguagens orientadas a procedimentos, existe uma diferença crucial: uma classe pode gerar coleções de objetos, sendo que cada objeto tem a sua memória privativa, determinando regras distintas de visibilidade (e, portanto, acesso) dos dados. Objetos que compartilham uma descrição de suas memórias privativas e que compartilham um conjunto comum de operações podem ser agrupados em uma classe [GOL83].

Vista assim, uma classe é um descritor de dados (e operações) que serão usados para a derivação de novas classes ou instanciados operacionalmente como objetos distintos quanto à sua memória privativa, enquanto que módulos permitem um compartilhamento de dados entre seus componentes (definidos localmente) baseado em alcance de identificadores, categorizados como globais ou locais. E módulos tradicionais representam ao mesmo tempo uma descrição e uma instância. Em resumo,

um módulo tradicional pode ser considerado equivalente a uma classe sem subclasses e com uma única instância.

O termo paradigma, empregado no seu sentido de modelo, padrão, entrou em voga na década de 80 para classificar as linguagens de programação de acordo com seu modelo conceitual de programação. O modelo de orientação a objetos tem continuamente originado novas linguagens de programação ou derivações de linguagens já existentes; a sua eficácia na solução de problemas gerais pode ser deduzida pela atual popularidade do modelo nas mais diversas áreas de aplicação e a sua eficiência, embora dependente de implementações particulares, não tem sido questionada de forma evidente na literatura. Seguindo o pensamento de Ambler [AMB92],

> Um paradigma de programação é uma coleção de padrões conceituais que, reunidos, moldam o processo de criação e, em última instância, determinam a estrutura do programa. Estes padrões conceituais estruturam o pensamento e assim determinam a forma dos programas válidos. Eles controlam como nós pensamos a respeito das soluções e sua formulação e, até mesmo, determinam se chegamos a alguma solução.

No contexto deste trabalho, duas linguagens merecem destaque especial, por razões distintas. Em primeiro lugar, a linguagem C++ [STRO91], por ser a escolhida para a formulação e implementação de protótipos, por sua:

- Flexibilidade: organização hierárquica de objetos, herança seletiva, herança múltipla, distinção entre objetos e grupos de objetos.
- Eficiência: código compilado, verificação estática de tipos.
- Disponibilidade: disponível em várias plataformas (DOS, UNIX), inclusive com extensões, como OpenC++ [CHI93a].
- Extensibilidade: facilidades de extensão por meio de bibliotecas de classes que separam a implementação das interfaces.
- Popularidade: na área de sistemas distribuídos orientados a objetos. Linguagem de implementação de Choices [CAM93] e Arjuna [SHR95].

Uma das críticas a C++ ser usada como linguagem de implementação de sistemas críticos é a fragilidade de sua linguagem base, a linguagem C. Esta crítica não tem rebate; portanto, as deficiências dessa linguagem devem

ser contornadas pelo uso cuidadoso de seus mecanismos mais suscetíveis a erro.

Em segundo lugar, a linguagem ADA, por sua concepção original de aplicabilidade na construção de sistemas de tempo real (que exigem alto grau de confiabilidade) e sua conhecida disciplina de desenvolvimento de programas. Acrescente-se ainda o seu forte relacionamento com o método HOOD - Hierarchical Object-Oriented Design usado para o projeto de classes de MOTF.

2.3 Justificativas

Neste ponto cabe uma pergunta: - Será o modelo de orientação a objetos tão fortemente distinto dos demais modelos de programação que justifique a individualização de técnicas como as de tolerância a falhas?

Para introduzir este assunto, é pertinente lembrar que linguagens de programação orientadas a objetos e, decorrentemente, o estilo de programação orientado a objetos tem se constituído em tópicos centrais de interesse, tanto a nível acadêmico quanto a nível de usuários não acadêmicos. Um exemplo recente é a extensão da linguagem ADA, denominada de ADA9X, que transforma esta linguagem *baseada* em objetos em uma linguagem *orientada* a objetos. Nesta extensão, a linguagem ADA passou a incluir um dos mais complexos e poderosos mecanismos do modelo de orientação a objetos: o mecanismo de herança. Sob o ponto de vista de desenvolvimento de programas, este mecanismo promove uma estruturação de programas completamente distinta de outros modelos e minimiza o esforço de programação. Mais adiante neste trabalho este mecanismo será revisto, para discutir a sua aplicabilidade no domínio de tolerância a falhas.

O projeto de programas segue o princípio destacado na própria denominação do modelo: orientação a objetos. O foco principal é o objeto, visto como um modelo físico, simulando o comportamento de uma parte real ou imaginária do mundo [KNU87]. Uma vez identificado o comportamento desejado, cada classe de objetos pode ser individualmente modelada, confinando e reduzindo a nível de uma classe a antes mencionada equação de Wirth.

Esta redução de complexidade a nível de classes de objetos torna peculiar este modelo para a construção de aplicações tolerantes a falhas. Primeiro, o comportamento tolerante a falhas pode ser especificado a um nível de granularidade menor (quando comparado a um módulo); segundo, o confinamento de dados e operações permite o isolamento de componentes críticos. Terceiro, a modelagem do comportamento de objetos por meio de classes abstratas pode ser concretizada através de implementações distintas, separando nitidamente os aspectos comportamentais e estruturais.

Tomando como base de comparação o estilo de programação orientado a procedimentos, no qual programas são estruturados obedecendo a uma hierarquia de módulos, o estilo de programação orientado a objetos pode ser visto como uma composição de objetos que interagem entre si, alguns solicitando e outros fornecendo serviços. Esta forma de estruturação de programas é dotada de um dinamismo que não encontra paralelo no modelo orientado a procedimentos, que é dotado de uma estrutura mais rígida. Objetos podem ser dinamicamente criados e multiplicados, sempre obedecendo a estrutura e o comportamento detalhado em sua classe. Dois objetos O_1 e O_2 instanciados da mesma classe C distinguem-se apenas pelo seu estado interno.

A construção de programas tolerantes a falhas pode se beneficiar deste dinamismo para a replicação e recuperação de componentes.

As interações entre os objetos realizam-se através de mensagens dirigidas a um alvo específico - um método de um objeto - mas nem sempre facilmente visível. Comparada a uma chamada de procedimento, uma mensagem apresenta peculiaridades que tornam a requisição de serviços mais suscetível a erros. Destacam-se as seguintes fontes de erros:

- Métodos virtuais: para permitir múltiplas redefinições de métodos, a ligação requisitante-requisitado é postergada da etapa da compilação para a etapa da execução. No momento da requisição, o método requisitado é selecionado, dependendo da objeto destinatário da chamada. Esta versatilidade pode implicar em ligações indesejáveis, não facilmente detectáveis pelo exame do texto do programa.

- Tratadores de mensagens: a liberdade de alteração estrutural oferecidas por algumas linguagens, a exemplo de SmallTalk, que permite a um objeto trocar de classe durante a execução [LIP93], exige a presença de um mecanismo de interpretação de

> mensagens com a finalidade de determinar precisamente onde se encontra o método fornecedor do serviço pretendido. Ou, em certos casos, rejeitar a mensagem por não conseguir estabelecer a ligação.

Entretanto, a versatilidade do mecanismo de mensagens, que se traduz em fragilidade como acima apontada, também proporciona a estrutura de implementação da reflexão computacional. Mensagens podem ser reinterpretadas ou redirecionadas para a realização de atividades que possam complementar ou mesmo modificar o comportamento especificado para o objeto destinatário da mensagem.

Considerando que o acima exposto justifica a interpretação das técnicas de tolerância a falhas para este modelo, outra questão se faz presente: - Este modelo é adequado à construção de software tolerante a falhas? Como ponto de partida, é interessante salientar duas colocações extraidas da seção Conclusões de [LIS94]:

a) As linguagens de programação imperativas sempre foram o paradigma do desenvolvimento de todas as técnicas, métodos, ferramentas e objeto de experimentos de tolerância a falhas e, mais especificamente, de mecanismos de tratamento de exceções. A norma MOD00-55, de 1991, do Ministério da Defesa do Reino Unido, conforme [PLA93], sugere fortemente que sejam usadas linguagens orientadas a procedimentos no desenvolvimento de sistemas de segurança crítica, ao invés de linguagens *assembler* ou outras linguagens "não convencionais".

b)Abbot [ABB90] advoga a idéia que uma linguagem de programação em lógica é ideal para a implementação de sistemas tolerantes a falhas, amparado nas seguintes razões básicas: mecanismo de retrocesso[1] que inclui, automaticamente, o salvamento/restauração de variáveis, que são características básicas da programação diversitária, especificações executáveis, que permitem ao programador trabalhar ao nível de especificação e mecanismos de implementação de programação diversitária de forma relativamente simples. A par destas facilidades, restrições podem ser apontadas, nas linguagens de programação em lógica, que limitam severamente a sua irrestrita aplicação em sistemas tolerantes a falhas. Desempenho, implementação segura de componentes encapsulados que

---

[1] *backtracking*.

possibilitem o confinamento de danos decorrentes de falhas, mecanismos poderosos de tratamento de exceções e estruturas de dados robustas, implementadas através de tipos abstratos de dados, não encontram apoio nas linguagens de programação em lógica. Tampouco o aspecto de controle de tempo em sistemas críticos de tempo real encontra facilidades diretas para a sua utilização.

2.4 Construção de objetos confiáveis

Nesta seção são estabelecidos os vínculos entre os domínios da aplicação e de tolerância a falhas. Não é intenção estabelecer um roteiro para o desenvolvimento de software crítico, mas apenas apontar as interseções entre os dois domínios, mostrando que a confiabilidade deve ser projetada a nível de objeto. Normas, padrões e sugestões para o desenvolvimento de software de segurança crítica podem ser encontradas em [PLA93].

2.4.1 Premissas

Nas premissas a seguir, um objeto é considerado o componente atômico da aplicação; falhas em suas propriedades são consideradas falhas provenientes de sua classe; erros e defeitos são confinados em objetos.

Premissa 1: *A confiabilidade de uma aplicação depende da confiabilidade de seus objetos (componentes).*

A confiabilidade é definida em termos de serviço continuado da aplicação, atendendo os seus requisitos. Como uma aplicação é decomposta em objetos e cada objeto é responsável por algum comportamento, certamente a confiabilidade de cada um dos objetos contribui para a confiabilidade global da aplicação e a negação também se aplica: caso um objeto não seja confiável, a confiabilidade da aplicação é comprometida.

Como argumento desfavorável a esta premissa, apresenta-se a perda de confiabilidade devido a *falhas de interface*. Estudos feitos por Basili e Perricone [BAS84] e Selby [SEL91] sobre programas modulares mostram que a interface entre os módulos é grande fonte de erros (39%), afetados pelas taxas de acoplamento/coesão entre eles: subsistemas com módulos de alto acoplamento/coesão apresentam 4.8 vezes mais erros por KNCSS (1000 No Commented Source Statements) do que subsistemas com baixas taxas de acoplamento/coesão.

Premissa 2: *Um objeto é a menor unidade de funcionalidade de uma aplicação.*

O modelo de orientação a objetos baseia-se em simulação comportamental de objetos do mundo real e agrupa objetos de comportamento similar em classes. Sob este aspecto, a classe define e implementa as funcionalidades desejáveis em um objeto e *nenhuma* outra mais que não seja diretamente relacionada com a definição de seu comportamento.

É inerente a este modelo a visão facetada de uma classe. Uma faceta é a definição de seu comportamento e outra faceta é a sua implementação. Sendo a primeira faceta bem definida, ela pode ser *abstrata* e permanecer *invariante*. A segunda faceta é a *concretização* de todo ou parte desse comportamento e é *variável*.

Na construção de uma aplicação, um objeto O_i pode ser *substituído* por um objeto O_k desde que a única diferença entre O_i e O_k seja no aspecto de implementação. Isto é, ambos possuem equivalência funcional, admitindo apenas diferenças em suas respectivas concretizações.

Premissa 3: *Um objeto é invulnerável.*

Cada objeto é o único responsável pelo seu estado interno. Nenhuma solicitação lícita de seus serviços poderá conduzir o objeto a um estado errôneo, corrompendo os seus dados ou seu código.

Ao encapsular e proteger seus dados de instância, somente o próprio objeto poderá alterá-los. Portanto, cabe ao objeto assegurar a sua confiabilidade.

2.4.2 Confiabilidade de um objeto

A confiabilidade de um objeto não se restringe a seus períodos de atividade como prestador de serviços. Um objeto torna-se ativo ao receber uma mensagem implícita ou explícita solicitando a execução de um de seus métodos.

Além dos métodos que definem e implementam os serviços do objeto, existem dois métodos que atuam sobre o próprio objeto e que podem ser ativados implicitamente: o construtor e o destruidor.

Ao método construtor são atribuídas responsabilidades tais como a instalação do objeto na memória e a inicialização de seus dados de instância, e, para isso, possui o poder de executar outros métodos do objeto, considerados privativos e não acessíveis a seus clientes.

Ao método destruidor é atribuída a responsabilidade de desativar o objeto, podendo executar outras atividades antes de destruí-lo.

Um objeto pode ter construtores, destruidores e outros métodos polimórficos que definem situações particulares de ativação.

Em decorrência das considerações acima, a confiabilidade de um objeto pode ser segmentada em três momentos: o momento da instanciação, o momento da prestação de serviços e o momento de sua destruição.

Continuando com essa linha de raciocínio, é possível que objetos instanciados a partir de uma mesma classe sejam expostos a situações de ativação completamente distintas, e, assim, suscetíveis a erros diversos. Resumindo, a obtenção de alta confiabilidade é uma atividade complexa no modelo de orientação a objetos.

2.4.3 Cenários

A seguir são abordados os diferentes cenários da vida de um objeto, desde o instante de sua concepção até a sua utilização como um componente de uma aplicação. Em cada cenário são sugeridas atividades pertinentes à prevenção, detecção e tolerância a falhas.

Cenário 1: Concepção com prevenção de falhas

Um objeto é uma abstração delineada a partir de seus requisitos funcionais - o que ele deve fazer - e não funcionais - o que ele deve observar.

Na especificação de requisitos funcionais, é importante delimitar a sua *abrangência*, objetivando a redução de seu tamanho (e de sua herança) e de sua complexidade. O mecanismo de herança incentiva a alta granularidade: herdar poucas e robustas propriedades promove a confiabilidade e reduz a complexidade. A herança múltipla exige cuidados especiais, devido à possibilidade de herdar métodos distintos que possuem a mesma interface.

No experimento de Selby e Basili [SEL91] sobre programação modular concluiu-se que o tamanho do módulo é estatisticamente significativo com respeito à taxa de erros detectados por milhar de comandos sem comentários (KNCSS - 1000 No Commented Source Statements) e esforço de reparação destes erros: módulos grandes (mais de 142 comandos) apresentam, em média, mais erros do que módulos pequenos (até 142 comandos) e o esforço correspondente para a sua remoção é cerca de duas vezes maior (5.08 horas/ KNCSS) do que em módulos pequenos (2.15 horas/KNCSS).

Métodos formais de especificação de requisitos são recomendados nesta fase, principalmente devido ao confinamento de comportamento propiciado pelo modelo. A validação da especificação pode ser auxiliada por especificações executáveis e complementada por outras técnicas. Jaffe e Levenson [JAF91] sugerem várias técnicas e critérios que auxiliam a descoberta de erros na especificação de requisitos, incluindo completude.

A especificação de requisitos *não funcionais* de um objeto é tarefa complexa, principalmente quando exige profundo conhecimento do domínio da aplicação ou, no caso oposto, não se relaciona diretamente com o domíno da aplicação exigindo conhecimento de outros domínios, como o de tolerância a falhas.

Cenário 2: Projeto com tolerância a falhas

Neste cenário a abstração começa a ser estruturada em termos de dados, interfaces e ambiente de execução. O projeto de tolerância a falhas deve ser feito neste momento, principalmente quando envolver projeto diversitário. A partir da especificação de requisitos são desenvolvidos projetos independentes que os atendam integralmente, com distintas abordagens.

Algumas técnicas de tolerância a falhas não se aplicam a componentes individuais, mas sim a grupos de objetos, como por exemplo, a técnica de conversações [XU95b,CLE93]. Outras técnicas podem ser individualizadas por objetos, ou ainda, podem ser definidas para métodos selecionados. Dependendo das exigências de confiabilidade do objeto, propriedades de tolerância a falhas podem ser incorporadas aos seus dados (ex. ações atômicas) ou a seus métodos (ex. redundância), de forma simples ou combinada.

Neste cenário existe a possibilidade de reuso. Classes abstratas podem ser concretizadas e classes concretas podem ser diretamente utilizadas ou usadas como base para um *projeto por diferença*; neste último caso, apenas as propriedades distintas são projetadas.

Cenário 3: Implementação

No modelo de orientação a objetos, a implementação não necessariamente gera um produto acabado, pronto para ser usado. A fase de implementação pode produzir diversos produtos, tais como classes básicas (funcionalidade reduzida), classes abstratas (com funcionalidades a serem concretizadas em classes descendentes) e classes parametrizadas (com propriedades variáveis).

Sob o ponto de vista de confiabilidade, todos os tipos de classe devem ter uma implementação *robusta*, incluindo mecanismos de auto-proteção de suas propriedades, não importando se a classe pode ser diretamente instanciada ou não. O mecanismo mais importante de auto-proteção é o mecanismo de *tratamento de exceções*. Através dele, a possibilidade de ocorrência de vários tipos de falhas pode ser antecipada, erros podem ser detectados durante o processo de execução e providências locais (internas ao objeto) podem ser tomadas com vistas à continuidade do serviço, ou mesmo, quando não é possível resolver internamente, o objeto pode sinalizar ao cliente de seus serviços a ocorrência de uma situação excepcional que o impede de fornecer o serviço solicitado.

A tolerância a falhas em software usando redundância de componentes exige nesta fase a implementação de objetos diversificados, derivados ou não da mesma classe ancestral, e a implementação de técnicas de gerenciamento de redundância. Técnicas de gerenciamento de redundância tem por objetivo controlar a execução de cada serviço solicitado a objetos replicados ou diversificados, fornecendo uma única resposta ao cliente desse serviço.

Cenário 4: Testes

Este cenário exige a presença de mais de um objeto: o cliente de seus serviços, ou o objeto servidor (quando o objeto testado é o cliente) ou ambos (quando o objeto testado é um agente - cliente e servidor). Embora a área de

testes seja uma área acadêmica de pesquisa, técnicas de tolerância a falhas podem contribuir para o diagnóstico de erros.

O diagnóstico de erros pode ser feito usando técnicas de *injeção de falhas* ou técnicas de programação diversitária, usadas separadamente ou combinadas. A primeira utiliza a abordagem sabotadora, tentando simular todas as situações de ocorrência de erros. A segunda faz testes baseados em similaridade de comportamento.

A seleção de objetos de mesmo comportamento externo, existentes em repositórios de objetos reusáveis, pode ser facilitada pelo uso de testes amparados nas técnicas de programação diversitária, que funcionariam como um "oráculo" para os casos de testes. Objetos de funcionalidade similar podem ser testados com grandes conjuntos de dados, sendo seus resultados submetidos a testes de aceitação ou votação; anotadas as suas diferenças, estas podem servir para a seleção do objeto mais adequado à aplicação (mesmo não sendo tolerante a falhas). Em relação a este aspecto, o trabalho de Kelly [KEL91] apresenta alguns dados interessantes sobre um experimento de implementação de projeto diversitario utilizado para testar objetos em sistemas distribuídos.

Cenário 5: Componentes da aplicação

Neste cenário, uma classe de objetos deve ser integrada à aplicação e interagir com os demais componentes. Sob o ponto de vista da aplicação, o único tipo de falha que pode se manifestar é aquela *existente* em um ou mais objetos ou na interação entre dois objetos (falha de interface).

E o domínio de tolerância a falhas oferece soluções que exigem a existência de código estranho ao domínio da aplicação, para fins de preservação do estado da computação, monitoração de comportamento de cada componente ou de grupos de componentes e gerenciamento de redundância.

2.5 Análise de características

A seguir serão analisadas as características mais marcantes do modelo de orientação a objetos e suas possibilidades de contribuição, tanto positivas quanto negativas, para a implementação de tolerância a falhas.

2.5.1 Processo de execução

Objetos são criados de forma dinâmica, durante o processo de execução, como novas instâncias de classes com estruturas de dados e comportamento bem definidos. O comportamento operacional de um objeto pode ser especificado através dos seguintes eventos:

- *Criação de um objeto:* uma nova instância de um objeto é criada, e seus dados de instância passam a ocupar seu espaço na memória.
- *Destruição de um objeto:* um objeto é destruído, liberando o seu espaço na memória.
- *Atribuição a um objeto:* os dados de instância de um objeto têm seus valores alterados.
- *Envio de mensagem:* uma mensagem parametrizada é enviada a um objeto destinatário, identificando o método desejado.
- *Ativação de um objeto:* o recebimento de uma mensagem desencadeia o início da execução de um método.
- *Desativação de um objeto:* o objeto termina a sua execução, mudando seu estado de ativo para passivo, mas continua a existir.

Neste momento salientam-se duas facetas do comportamento operacional: a criação de novos objetos por instanciação de classes e a atribuição de objetos. Enquanto a primeira é uma forma natural de replicação de componentes, a segunda propicia a preservação de estado por cópia de todo o componente mesmo sem saber detalhes de sua descrição. Estas facetas são únicas deste modelo e, se comparado ao modelo orientado a procedimentos, estas questões podem ser assim colocadas: (a) como seria possível replicar de forma dinâmica um módulo de um programa? (b) como seria possível copiar todas as variáveis de estado de um procedimento desconhecendo a sua estrutura?

2.5.2 A complexidade

Algumas características de linguagens orientadas a objetos, tais como sobrecarga de funções e operadores adicionam complexidade ao processo de execução de programas. O código executável de um objeto possui uma estrutura dinâmica, pois o polimorfismo exige a amarração de alguns nomes em tempo de execução, dificultando a verificação e validação de programas. Além disto, a sobrecarga de funções e operadores determina por

contexto semântico qual objeto está sendo referido, pois seu nome não é suficiente para uma identificação unívoca.

Uma exploração positiva de tolerância a falhas em relação ao aspecto de amarração dinâmica[2] é a possibilidade de substituição dinâmica de componentes, promovendo a mudança de implementação e mantendo estáveis as referências a estes componentes.

Outro nível de complexidade é adicionado ao comportamento de objetos se for contemplada a possibilidade de herança dinâmica. A herança de classe é essencialmente estática, visto que as propriedades herdadas são precisamente definidas no texto fonte e permanecem inalteradas durante o processo de execução. Já a *herança dinâmica* é passível de duas interpretações:

a) refere-se a mecanismos que permitem a objetos alterar o seu comportamento em tempo de execução, seja aceitando novas propriedades de outros objetos, seja por mudanças em seu meio ambiente [NIE89]. No sistema operacional Apertos, a mudança de contexto é realizada em um meta-nível, por migrações de objetos em meta-espaços.

b) o polimorfismo também pode ser visto como herança dinâmica no momento em que o comportamento de um método depende de seu contexto.

Sob o ângulo do texto fonte, a complexidade também é maior, se comparada a linguagens orientadas a procedimentos. A herança, simples ou múltipla, conduz a uma distribuição sintática da descrição completa do código e dos dados que compõem uma instância de um objeto. Enquanto semanticamente dependente das classes ancestrais, o objeto tem seu código sintaticamente distribuído ao longo do texto fonte, não observando o conceito de *localidade* de ações.

2.5.3 Execução concorrente, paralela e distribuída

Na execução concorrente, diversos objetos podem estar ativos em um determinado instante e cooperando entre si para a consecução de alguma tarefa ou competindo entre si para a obtenção de algum recurso, de forma sincronizada.

---

[2] *dynamic binding*.

As técnicas de tolerância a falhas exploram a concorrência para evitar a degradação de desempenho do sistema no caso de utilização de redundância ativa e para replicação de componentes.

Relembrando aqui que, no caso de execução distribuída de objetos, ambientes de suporte tem sido desenvolvidos com características de tolerância a falhas, evidencia-se a importância do uso de técnicas de tolerância a falhas em aplicações distribuídas orientadas a objetos. Shrivastava [SHR95], menciona que o projeto e a implementação de Arjuna teve como objetivo propiciar o estado da arte em programação de sistemas para a construção dessas aplicações.

Aplicações de tempo real tolerantes a falhas no modelo de orientação a objetos também se constituem em tópicos atuais de pesquisa. Kim [KIM95] preocupa-se com a integração de técnicas de tolerância a falhas em tempo real estruturadas com base em processos, com a próxima geração de técnicas de estruturação de sistemas de tempo real orientados a objetos. Salienta a importância desta integração, considerando os benefícios de modularidade, generalidade e abstração natural que o modelo proporciona.

2.5.4 Encapsulamento

Sob o ponto de vista de tolerância a falhas, o encapsulamento apresenta visíveis vantagens, principalmente no que se refere à detecção de erros, na forma de comportamentos anômalos e de recuperação de erros, restringindo a possibilidade de contaminação de outros objetos do sistema. Ao esconder a representação interna do objeto e tornar visível apenas a sua interface, aumenta a facilidade de diversificação de implementação de componentes com permanência de interface.

2.5.5 Herança

A herança estática, que transmite integralmente ou seletivamente propriedades na hierarquia de classes, pode ser explorada no sentido de estender aspectos comportamentais de tolerância a falhas na hierarquia de objetos do sistema de várias formas:

a) Diversidade de implementações: operações do domínio de tolerância a falhas podem ser definidas sobre uma classe base. A classe base

abstrata define a interface única a partir da qual são feitas as diversas implementações dos métodos críticos.

b) Especialização de comportamentos: metaobjetos herdam os protocolos básicos de reflexão computacional de classes especiais e os especializam; classes de meta-objetos utilizam o mecanismo de herança para particularizar comportamentos.

Superclasses e subclasses permitem a fatoração do sistema, minimizando repetições das mesmas operações em diferentes lugares. A habilidade de herdar propriedades, de forma integral ou seletiva, a possibilidade de aumentar, modificar ou anular esta herança, viabiliza a programação por diferença [GOL83]. Apenas as propriedades distintas necessitam ser implementadas, mantendo intocadas as demais propriedades. Uma conseqüência imediata da programação por diferença é a manutenção da confiabilidade dos componentes já existentes, verificados e/ou testados pelo uso.

2.5.6 Polimorfismo

Uma *mensagem* enviada a um objeto provoca um determinado comportamento no objeto receptor e, se a mesma mensagem for enviada a diferentes objetos, o comportamento poderá ser distinto. A esta interpretação particular de cada objeto frente à mesma mensagem, dá-se o nome de *polimorfismo* .

Este mecanismo que se traduz na forma de métodos virtuais também contribui para a particularização das técnicas clássicas de tolerância a falhas para este modelo. Por exemplo, algoritmos de decisão e aceitação, um dos pontos de apoio das técnicas de programação diversitária, podem assumir comportamentos distintos ditados por seus argumentos, enquanto homônimos.

Críticas sobre o polimorfismo apontam a dificuldade de verificação de programas com amarração dinâmica de componentes e seu custo associado. No relatório que descreve o projeto de extensão [DoD93] da linguagem Ada83 (baseada em objetos) para Ada9x (orientada a objetos) a adoção de amarração dinâmica nesta última é assim justificada: "... a ausência de amarração dinâmica em Ada83 foi inadequada. Agora sabe-se que os custos de implementação são triviais e irrelevantes; a verificação não

é um argumento relevante, pois agora é sabido que em qualquer software de segurança crítica onde a verificação é uma necessidade, somente é usado um pequeno subconjunto da linguagem."

2.6 Reutilização

Linguagens orientadas a objetos renovam e enfatizam o conceito de reutilização de código, sob forma de classes de objetos genéricos mantidos em bibliotecas e selecionados por critérios de funcionalidade e adequação à aplicação em desenvolvimento.

Este conceito pode ser aproveitado para o desenvolvimento de classes de objetos que implementem funcionalidades de tolerância a falhas de forma padronizada e totalmente reusável, bem como para promover a reutilização dos próprios objetos da aplicação. Objetos sem características de tolerância a falhas podem ser reutilizados para a construção de aplicações tolerantes a falhas.

A programação por diferença, acima associada ao mecanismo de herança, também pode ser realizada por parametrização de classes, que permitem especificar *classes genéricas*. Classes genéricas[3] são modelos de classes cujos tipos de dados ainda estão em aberto. Essas classes, assim como as *classes abstratas*, não são instanciáveis; elas devem ter seus parâmetros definidos para a sua completa concretização. Já as classes abstratas devem possuir subclasses onde são feitas as concretizações das abstrações herdadas.

A técnica de re-expressão de dados[4] encontra nas classes parametrizadas uma aplicação direta de programação por diferença de dados. Esta técnica admite e re-execução do mesmo código sobre o mesmo conteúdo de informação, codificado sob diferentes formatos de dados; desta forma, alternativas de funções críticas podem ser geradas com base nas mesmas operações, sobre dados de diferentes tipos ou domínios.

2.7 Comentários

Neste capítulo foram colocadas duas questões como base de referência: (a) Será o modelo de orientação a objetos tão fortemente distinto

---

[3] *templates*.

[4] Vide Cap.1, seção 1.4.1.

dos demais modelos de programação que justifique a individualização de técnicas como as de tolerância a falhas? ; (b) Este modelo é adequado à construção de software tolerante a falhas?

Discorrendo sobre estes temas, por diferentes lados, ambas podem ser respondidas afirmativamente.

A *individualidade* do modelo tem como característica mais marcante o mecanismo de herança. Sob o ponto de vista de desenvolvimento de programas, este mecanismo promove uma estruturação de programas completamente distinta de outros modelos, seguindo o princípio destacado na própria denominação do modelo: orientação a objetos.

Classes são modeladas como descritores de dados e operações, que constituem as denominadas propriedades da classe. Propriedades que podem ser transmitidas de forma integral ou seletiva a subclasses; propriedades que podem ser usadas, substituídas ou ignoradas pelas classes descendentes; propriedades que são confinadas a uma classe de objetos, não sendo acessíveis àqueles que não pertencem à classe. Classes de objetos tornam peculiar este modelo para a construção de aplicações tolerantes a falhas.

O comportamento tolerante a falhas pode ser especificado e confinado a um nível baixo de *granularidade* menor e o confinamento de dados e operações permite o *isolamento* de componentes críticos. A modelagem do comportamento de objetos por meio de classes abstratas pode ser concretizada através de implementações distintas, separando nitidamente os aspectos comportamentais e estruturais.

Objetos podem ser dinamicamente criados e multiplicados, sempre obedecendo a estrutura e o comportamento detalhado em sua classe; a construção de programas tolerantes a falhas pode se beneficiar deste dinamismo para a replicação e recuperação de componentes.

O modelo de orientação a objetos também proporciona a estrutura de implementação de *reflexão computacional*. Mensagens podem ser reinterpretadas ou redirecionadas para a realização de atividades que possam complementar ou mesmo modificar o comportamento especificado para o objeto destinatário da mensagem. A reflexão computacional no modelo de orientação a objetos é apresentada no capítulo 3 e sua

interpretação para o domínio de tolerância a falhas é apresentada no capítulo 4.

A *adequação* do modelo, explicitada ao longo deste trabalho, também já foi percebida por diversos pesquisadores. Na Universidade de Newcastle upon Tyne, berço da técnica de blocos de recuperação e importante centro de pesquisa em aplicações de alta confiabilidade, vários pesquisadores tem se dedicado a diversos aspectos de tolerância a falhas no modelo de orientação a objetos [SHR95], [SHR94], [RUB94], [XU95], [FAB95].

O desafio e a atualidade do tema são evidenciados, não apenas pelas recentes pesquisas na área, mas também por questões ainda sem solução consolidada. Kim [KIM95] e seu grupo da Universidade da California, Irvine, USA, propalam que a nova geração de sistemas de tempo real é estruturada de acordo com o modelo de orientação a objetos, e que a integração das técnicas de tolerância a falhas atualmente existentes a essa nova estruturação ainda requer esforços intensivos de pesquisa analítica e experimental.

Capítulo 3

REFLEXÃO COMPUTACIONAL

A reflexão computacional no modelo de orientação a objetos é apresentada em suas diversas interpretações e sintetizada através de definições. A síntese capta o enfoque de orientação a objetos e fornece a base para a descrição de objetos reflexivos.

3.1 Introdução

Os múltiplos significados do termo reflexão[1] exigem um cunho específico no contexto deste trabalho: tanto reflexão, como ato introspectivo de meditação ou conclusão dela advinda, quanto reflexão como propriedade física modificação da direção de propagação de luz, calor, som, desviando da primeira direção, se adequam para definir o significado de reflexão computacional. O primeiro significado traduz o objetivo de reflexão computacional, que é atuar sobre si mesmo, fazendo computações sobre dados do próprio sistema. Já o segundo significado relaciona-se com a implementação de reflexão computacional no modelo de orientação a objetos, como será discutido a seguir. No conceito de Maes [MAE87], reflexão computacional é a atividade executada por um sistema computacional quando faz computações sobre (e possivelmente afetando) suas próprias computações.

Neste conceito, a reflexão é definida como uma forma de meditação, no qual o sistema tenta tirar conclusões sobre as suas próprias computações, as quais podem ser afetadas posteriormente. Uma definição mais detalhada é dada por Steel [STE94]:

> A reflexão computacional é a capacidade de um sistema computacional de interromper o processo de execução (por exemplo, quando ocorre um erro), realizar computações ou fazer deduções no meta-nível e retornar ao nível de execução traduzindo o impacto das decisões, para então retomar o processo de execução .

Nesta última definição é introduzido o conceito de níveis de computação, aproximando-se do significado físico de reflexão, visto que a ocorrência de um evento (por exemplo, um erro) no processo de execução reflete-se em um nível superior (meta-nível) de computação. Em resumo, por **reflexão computacional** entende-se:

> Def. 3.1: **Reflexão computacional** é toda a atividade de um sistema computacional realizada sobre si mesmo, e de forma separada das computações em curso, com o objetivo de resolver seus próprios problemas e obter informações sobre suas computações.

A reflexão computacional define uma arquitetura em níveis, denominada **arquitetura reflexiva**, composta por um meta-nível, onde se

---

[1] Fonte: Novo Dicionário da língua Portuguesa, Aurélio B. H. Ferreira, Ed. Nova Fronteira, 1975.

encontram as estruturas de dados e as ações a serem realizadas sobre o sistema objeto, localizado no nível base. Assim, em uma arquitetura reflexiva, um sistema computacional possui dois componentes interrelacionados: um subsistema objeto, que faz computações sobre um domínio externo ao sistema, e um subsistema reflexivo, que faz computações sobre o sistema objeto, conforme esquematizado na figura 3.1. Os dados do nível base são usados no meta-nível para a realização de computações reflexivas que podem interferir nas computações de nível base.

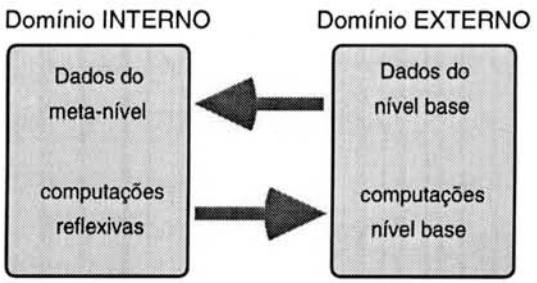


Figura 3.1 - Arquitetura reflexiva

Por domínio de um sistema computacional entende-se todo conteúdo de informação, expresso por objetos e operações, relacionado com uma determinada área. Para efeito das definições a seguir, designa-se de domínio *externo* do sistema ou *domínio da aplicação* o conteúdo de informação relacionado com a aplicação, que é a abstração de mais alto nível, e por domínio *interno* do sistema ou auto-domínio o conteúdo de informação sobre o próprio sistema, incluindo a representação interna do domínio da aplicação.

> Def. 3.2: **Sistema reflexivo** é um sistema computacional que possui um componente capaz de realizar computações sobre o domínio interno do sistema.

O componente reflexivo pode atuar sobre a estrutura ou sobre o comportamento do sistema objeto. No primeiro caso, o meta-nível é composto por **meta-classes**, as quais contém informações sobre os aspectos estruturais do nível base: descrição de variáveis e de métodos que irão compor todas as suas instâncias. Esta estrutura pode ser alterada e, em decorrência disso, todas as instâncias também serão alteradas. No segundo caso, o meta-nível é composto por **meta-objetos**, que contém informações sobre os aspectos de comportamento dos objetos (instâncias de classes) de nível base, como por exemplo, como um determinado objeto trata uma

mensagem. As seções 3.2 e 3.4 descrevem mais detalhadamente estes conceitos.

3.2 Meta-classes

Algumas linguagens orientadas a objetos utilizam meta-classes para descrever a estrutura de todas as suas classes, no sentido de que as informações necessárias para a construção de uma classe estão estruturadas em sua meta-classe. Entre estas informações encontra-se o nome da classe, a descrição de suas variáveis e informações sobre herança, entre outras. Mais especificamente, meta-classes podem ser assim definidas, com base em [BOO94], [GRA89]:

> Def. 3.3: Uma **meta-classe** $\mathcal{M}c_{\chi}$ é uma classe que descreve a estrutura de uma classe χ e cujas instâncias também são classes.

A propriedade de ser instanciável como uma nova classe permite que classes possam ser diretamente manipuladas, como se fossem objetos. E também as distinguem de classes tradicionais (que descrevem a estrutura de objetos), visto que o seu domínio é a própria classe; seus dados referem-se a nomes de métodos, instâncias, heranças e seus métodos obtém informações e fazem alterações sobre os seus dados.

Em linguagens que estruturam as suas classes a partir de meta-classes, estas propriedades são herdadas pelas classes, de forma que todas as classes são especializações de suas meta-classes. Na árvore de herança, a meta-classe é a super-classe de todas as classes e, conseqüentemente, as classes de todos os objetos são descendentes diretas de alguma meta-classe. Isto significa que a alteração em alguma meta-classe da árvore de herança pode ser percebida por todos os descendentes da árvore de herança.

3.3 Reflexão estrutural de classes

Meta-classes, quando acessíveis aos usuário, permitem a realização de reflexão estrutural, que pode ser assim definida, com base em [FER89] :

> Def. 3.4: **Reflexão estrutural** de uma classe χ é toda a atividade realizada em uma meta-classe $\mathcal{M}c_{\chi}$, com o objetivo de obter informações e realizar transformações sobre a estrutura estática da classe χ.

Informações e transformações típicas de reflexão estrutural são: (a) obter informações sobre a classe χ: sua classe ascendente, suas descendentes,

suas instâncias, seus métodos e interfaces; (b) alterar a classe x: modificar suas variáveis e seus métodos; e ainda, (c) atuar sobre classes: criar novas classes, eliminar classes existentes e renomear classes.

A título de exemplo, todas as classes de SmallTalk são instâncias de uma meta-classe OBJECT e as facilidades reflexivas são implementadas em meta-classes especiais da hierarquia. As principais meta-classes de Smalltalk, estão resumidas no quadro a seguir, compilado com base em [LIP93]:

Tabela 3.1 - Meta-classes de Smalltalk

Denominação da classe e → classe ancestral	**Atua sobre**	**Dados reflexivos**
Object	classes e meta-classes	nomes de mensagens
Behavior → Object	classes e objetos	nome da classe, hierarquia, nome da tabela de métodos, descrição das instâncias e estado p/interpretador e compilador
Class → Behavior	classes	representação das variáveis da classe e variáveis compartilhadas
Metaclass → Behavior	meta-classes	protocolo de inicialização de variáveis da classe

A computação reflexiva estrutural de classes atua sobre todas as instâncias da classe. Por exemplo, quando o tratador de mensagens da meta-classe é alterado, o objeto recebe uma mensagem e a envia para a sua meta-classe; esta possui um interpretador de mensagens genérico, compartilhado com todas as instâncias da mesma classe.

Quando uma mensagem é dirigida a um objeto, é feita uma pesquisa no dicionário de mensagens, que é parte da representação de sua classe, e contém todos os métodos aos quais as instâncias desta classe podem responder [BOO94]. Se esta representação é alterada, todos os objetos sofrerão a mesma alteração em seu tratamento de mensagens e, portanto, a reflexão estrutural difere da reflexão computacional, no sentido de ser esta última particularizada por objeto e adequada para atividades tais como registro e monitoração do comportamento do objeto.

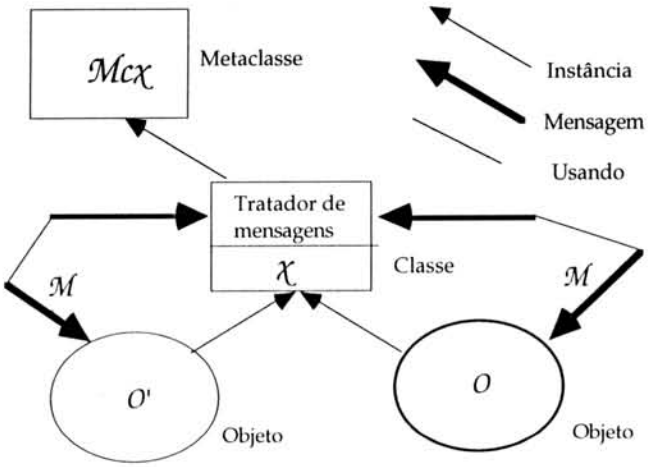


Figura 3.2 - Meta-classes

A figura 3.2 esquematiza uma árvore de instanciação, com dois objetos terminais e o tratamento de suas mensagens.

3.4 Meta-objetos

Genericamente, o conceito de meta-objeto é assim colocado por Foote [FOO92]:

Meta-objetos (ou objetos de meta-nível) são objetos que definem, implementam, dão suporte ou participam de qualquer maneira da execução da aplicação, ou objetos de nível base .

Em linguagens orientadas a objetos, o primeiro modelo de reflexão computacional foi introduzido por Maes [MAE87], tendo como base o emprego de *meta-objetos*, com as seguintes características:

- cada objeto na linguagem possui um meta-objeto, em uma relação um-para-um;
- as estruturas contidas no meta-objeto do objeto (seu *referente*)[2], contém **todas** as informações sobre a implementação e a interpretação do objeto;
- Um objeto pode interromper a sua execução, refletir sobre si mesmo (como representado em seu meta-objeto) e modificar seu comportamento futuro;
- os meta-objetos, por sua vez, são considerados objetos, podendo ter seus próprios meta-objetos.

No modelo de Maes, implementado no interpretador da linguagem 3-KRS, um conjunto de meta-objetos são usados como primitivas para

---

[2] Um objeto que uma descrição representa é denominado de *referente* da descrição [Stee88]

representar o interpretador e os objetos da linguagem. Portanto, quando um objeto é criado em alguma aplicação, ele automaticamente herda de algum destes meta-objetos, de acordo com o seu tipo. Uma variação do modelo de Maes é posteriormente introduzida por Ferber [FER89] considerando meta-objetos como instâncias de uma classe chamada Meta-objeto, que se distingue da classe de seus referentes. A classe do objeto encapsula a descrição estrutural (definição da estrutura de suas instâncias e o conjunto de métodos) e o seu meta-objeto descreve seus aspectos computacionais (como uma mensagem é interpretada e um método é aplicado). Das colocações acima destacam-se como importantes no contexto deste trabalho as seguintes características: um meta-objeto é um objeto instanciado a partir de uma classe, este objeto descreve alguns aspectos (não necessariamente todos) de um outro objeto ao qual se refere, e participa do processo de execução de seu objeto referente. Em síntese, um meta-objeto pode ser assim definido:

> Def. 3.5: Um **meta-objeto** $Mo_{\mathcal{X}}$ é um objeto que representa aspectos estruturais e comportamentais de um objeto O, a ele conectado. A estrutura de O é representada como dados de $Mo_{\mathcal{X}}$ e os aspectos computacionais são descritos como métodos de $Mo_{\mathcal{X}}$.

A figura 3.3 esquematiza a conexão de um objeto O a seu objeto $Mo_{\mathcal{X}}$, pertencentes a classes distintas. Mensagens enviadas ao objeto são dirigidas a seu meta-objeto.

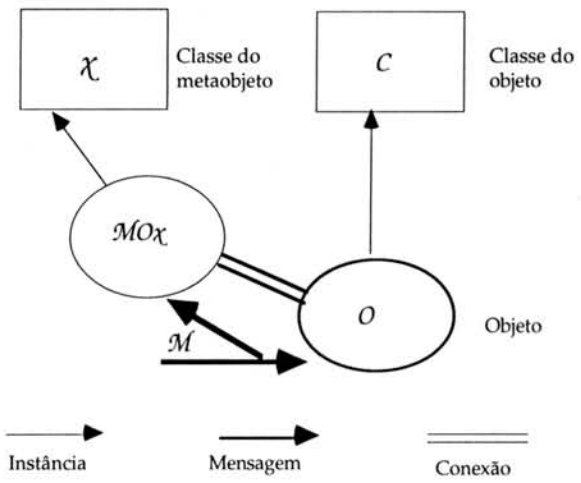


Figura 3.3 - Meta-objetos

Na definição de uma arquitetura reflexiva, Ferber [FER89], destaca as seguintes questões:

i) quais entidades devem ser transformadas em algo que possa ser manipulado no meta-nível?

ii) como o relacionamento entre o nível base e o meta-nível é implementado?

iii) quando o sistema passa para o meta-nível?

A primeira questão refere-se aos dados que podem ser manipulados no meta-nível. A atividade computacional do nível base é transformada em dados para o nível superior, determinando quais computações podem ser realizadas. Esta operação de transformação é denominada de reificação [FER89], [NAK92], assim definida:

Def. 3.6: **Reificação** é a transformação da execução de objetos em dados que possam sofrer operações no meta-nível.

A segunda e a terceira questões indagam sobre quem e como inicia o processo de reflexão. Maes[MAE88] aponta duas soluções: a responsabilidade pode ser atribuída ao objeto de nível base, que neste caso contém código mencionando seu meta-objeto ou pode ser atribuída ao sistema (interpretador). Neste último caso, as mensagens enviadas ao objeto e que iniciam sua atividade computacional, são delegadas ao seu meta-objeto, passando este a ser o responsável pelo tratamento da mensagem.

3.5 Reflexão comportamental de objetos

Na reflexão comportamental de objetos, também conhecida como reflexão computacional, a função básica do meta-objeto é explicitar como o objeto reage diante de uma mensagem, possibilitando a intervenção no estado da computação. Esta intervenção é a essência da computação reflexiva dinâmica, buscando coletar e registrar informações sobre o processo de execução, a exemplo de estatísticas sobre desempenho, informações para fins de depuração e monitoração da execução, ou com a finalidade de modificar o curso do processo de execução, como por exemplo determinar qual computação deve ser feita a seguir, ativar computações alternativas (através de monitores e *daemons*).

Def. 3.7: **Reflexão comportamental** de um objeto o é toda a atividade realizada por um meta-objeto $\mathcal{M}o_{\chi}$ com o objetivo de obter informações e realizar transformações sobre o comportamento do objeto o.

No contexto deste trabalho, os objetos controlados por meta-objetos são denominados **objetos reflexivos**. Para realizar a reflexão comportamental de objetos, são necessárias poucas informações sobre a sua estrutura dos objetos [MAL92] ; acesso à informações estruturais sobre objetos individuais e reificação de métodos na forma de objetos são suficientes. Seguindo esta linha de reflexão, pode-se afirmar que: (a) é possível obter distintas granularidades de reflexão comportamental; (b) a reflexão comportamental não interfere no encapsulamento do objeto.

Como o comportamento de um objeto é ditado por seus métodos, a reificação individualizada de uma mensagem que ativa um método e seu conseqüente tratamento pelo meta-objeto, faz com que a reflexão seja restrita ao método chamado, ou seja, torna-se uma reflexão computacional particular de cada método e não de todo o objeto. Os métodos de um objeto reflexivo são divididos em dois grupos, os reflexivos e os não-reflexivos, onde somente os primeiros tem a sua execução controlada pelo meta-objeto.

A propriedade de encapsulamento é essencialmente estática e assegura que um objeto seja visível apenas por suas interfaces, tornando invisível a sua implementação interna (suas variáveis de instância e seus métodos). Na reflexão comportamental realizada por interceptação de mensagens, esta propriedade não é alterada, visto que a estrutura interna do objeto permanece inviolada; apenas aspectos de seu comportamento podem ser substituídos externamente. Ao contrário da reflexão comportamental, a reflexão estrutural viola o encapsulamento do objeto, visto que pode substituir seus métodos por outros métodos, de forma temporária ou permanente.

Meta-objetos, ao serem considerados objetos, podem enviar e receber mensagens de outros meta-objetos. Quando um meta-objeto recebe diretamente mensagens de outros meta-objetos, realiza-se uma computação reflexiva do sistema. Esta é a abordagem adotada por 3-KRS [MAE87] e ABCL/R [WAT88].

3.6 Torre de reflexão

A arquitetura reflexiva admite diversos meta-níveis, caracterizando uma *torre de reflexão*, na terminologia de [FER89]. Cada nível da torre de reflexão constitui um domínio D_i, considerado como domínio base do domínio D_{i+1}, seu meta-domínio situado no meta-nível imediatamente

superior. Cada domínio D_i é, ao mesmo tempo, o domínio base do domínio do nível superior D_{i+1}, e o meta-domínio do domínio D_{i-1}, situado no nível inferior, exceto por D_0, constituído por referentes que podem ser usados apenas no nível base [ANC95].

A monitoração de objetos em tempo de execução é descrita por Watanabe [WAT88], utilizando dois níveis de meta-objetos na linguagem ABCL/R. Seja χ o objeto monitorado e Mo_χ o seu meta-objeto. Seja Monitor um objeto que faz a monitoração das mensagens de χ, da seguinte forma: quando χ recebe uma mensagem m, o objeto Monitor deve ser informado por χ.

Quando o objeto Monitor recebe, dinamicamente, a incumbência de monitorar o objeto χ ele deve modificar o meta-objeto Mo_χ para que este passe a fazer a reflexão das mensagens para o Monitor. Esta modificação é feita através de um meta-objeto de Mo_χ denominado M_Mo_χ que adiciona um novo comportamento (método ou *script*) ao meta-objeto Mo_χ Este *framework* de monitoração de objetos utiliza a criação tardia (*lazy*) de meta-objetos para implementar a definição circular de meta-objetos, de forma a evitar uma torre infinita de meta-objetos. Esta torre infinita é induzida pelo fato que cada meta-objeto Mo_χ é considerado como um objeto, podendo ter seu próprio meta-objeto M_Mo_χ que implementa Mo_χ da mesma forma que Mo_χ implementa o objeto χ, e assim sucessivamente.

3.7 Meta-objetos como agentes de extensão

A partir do trabalho de Maes, que apontou a reflexão como uma característica intrínseca de linguagens orientadas a objetos, a reflexão computacional passou a ser considerada um importante mecanismo de extensão estática e dinâmica de linguagens orientadas a objetos.

Linguagens de programação, por concepção, possuem uma semântica bem definida, a ser obedecida pelos seus usuários no desenvolvimento de aplicações. A reflexão computacional muda esta ideia, ao permitir alterações na implementação da linguagem. Por exemplo, a alteração da classe de um objeto flexibiliza o conceito de que objetos herdam estaticamente a estrutura de sua classe ancestral. Ou ainda, ao interceptar uma mensagem dirigida a um objeto e reenviá-la ao seu meta-objeto, flexibiliza o conceito de que o objeto, ao receber uma determinada mensagem tem o comportamento ditado pelo método ativado por tal

mensagem. Esta flexibilização tem sido implementada em linguagens de programação através de um *protocolo de meta-ojetos* (MOP - metaobject protocol)[KIC91], que permita operações com meta-objetos.

3.8 Considerações finais

Para fazer computações sobre o sistema objeto, o componente reflexivo deve ser a ele conectado, de forma temporária ou permanente. A conexão temporária tem comportamento similar a um *daemon*, ou seja, um componente que monitora um objeto e que é ativado por alguma ação especial, como por exemplo, uma mudança de estado no objeto controlado.

A diferença essencial entre meta-objetos e *daemons* reside na própria reflexão, que permite alterar o comportamento do objeto base, por exemplo, adicionando a ele novas funcionalidades. Com a conexão temporária também é possível estabelecer novas conexões entre objetos de nível base e meta-objetos, permitindo que objetos de nível base adquiram funcionalidades distintas, a exemplo da arquitetura adotada no sistema operacional Apertos [YOK92], na qual os objetos são associados a meta-espaços, com diferentes funcionalidades.

A conexão permanente é estabelecida de forma estática, ligando um objeto a um meta-objeto. Sempre que o objeto receber uma mensagem, o meta-objeto a intercepta, faz a reificação e realiza as computações no meta-nível. Este é o modelo adotado em OpenC++, com algumas variações; o programador pode escolher métodos e variáveis de classe que devem ser reflexivos, permanecendo inalterados os demais.

Neste trabalho o modelo de reflexão computacional adotado é o de conexão estática, onde objetos e meta-objetos são associados em tempo de compilação. A próxima seção aborda a utilização de meta-objetos para atividades de tolerância a falhas, na interpretação do framework MOTF-Meta-Objetos para Tolerância a Falhas.

Capítulo 4

META-OBJETOS PARA TOLERÂNCIA A FALHAS

Apresenta o *framework* MOTF- Meta-Objetos para Tolerância a Falhas para a construção de aplicações tolerantes a falhas. Mostra exemplos e descreve a sua aplicabilidade em técnicas clássicas de tolerância a falhas em software.

4.1 Introdução

Neste capítulo são sintetizados os componentes do *framework* MOTF- Meta-Objetos para Tolerância a Falhas, a saber:

Domínio: **Tolerância a Falhas**

Modelo: **Orientação a Objetos**

Arquitetura: **Reflexão Computacional**

O primeiro componente define o foco de interesse, o segundo define a abordagem estrutural e o terceiro identifica a forma de implementação.

A área de tolerância a falhas abrange uma série de técnicas com funcionalidades e aplicabilidades bem definidas, permitindo que seja considerada um domínio em si. Ao conjugar o domínio de tolerância a falhas a um outro domínio, ou seja, ao domínio de uma aplicação, o primeiro passa a ser responsável pelos requisitos *não funcionais* da aplicação. Os requisitos não funcionais de uma aplicação, a exemplo de confiabilidade e segurança, são cruciais em muitas aplicações e exigem métodos e conhecimentos que são distintos do domínio da aplicação.

No modelo de orientação a objetos, uma aplicação é estruturada a partir de seus componentes atômicos: os objetos. Um objeto é uma abstração delineada a partir de seus requisitos funcionais - o que ele deve fazer - e não funcionais - o que ele deve respeitar. Sob esta ótica, um objeto possui duas facetas: a sua faceta operacional, como prestador ou solicitante de serviço atendendo às funcionalidades definidas, e sua faceta de confiabilidade, que pode ser observada pela constância e correção de fornecimento de seus serviços.

O *framework* MOTF baseia-se nesta visão facetada: objetos do domínio da aplicação possuem a responsabilidade de observar os requisitos funcionais da aplicação, de forma confiável, e objetos do domínio de tolerância a falhas buscam assegurar alta confiabilidade através de técnicas de tolerância a falhas, fornecendo e gerenciando a redundância de objetos do domínio da aplicação.

Numa abordagem inicial de requisitos, um objeto do domínio de tolerância a falhas deve atender às exigências de:

a) *Confiabilidade*: a sua execução deve ser confiável, visto que a execução correta de qualquer dos objetos do domínio da aplicação depende da execução correta dos objetos do domínio do tolerância a falhas.

b) *Separação semântica*: enquanto cada objeto isolado da aplicação possui sua própria semântica, no momento em que interage com o objeto do domínio de tolerância a falhas, uma nova semântica é adicionada ao objeto. Por exemplo, um objeto responsável pela funcionalidade f(x), ao se tornar um participante da técnica de programação em n-versões, f(x) torna-se apenas um dos resultados dentre um conjunto de resultados fornecidos pelos objetos participantes.

c) *Modularização*: os mecanismos de tolerância a falhas devem ser separadamente implementados pelos objetos, de forma a permitir a seleção e composição dos mecanismos mais adequados a cada aplicação.

d) *Composição*: propriedade intrínseca do modelo de orientação a objetos, a possibilidade de composição dos diversos mecanismos de tolerância a falhas e dos objetos da aplicação deve ser respeitada, no mais alto grau possível.

e) *Especificação de comportamento de alto nível:* os objetos do domínio de tolerância a falhas devem possuir um comportamento observável bem definido e que independam de características particulares de linguagens orientadas a objetos e seu sistema de suporte.

Em resumo, estes requisitos enfatizam a necessidade de separação dos domínios da aplicação e de tolerância a falhas e sugerem que as funcionalidades de tolerância a falhas sejam adicionadas aos objetos da aplicação evitando a necessidade de alteração estrutural destes últimos.

A separação de funcionalidades pode ser abordada sob diferentes formas e técnicas de implementação.

A abordagem de RMP - Recovery Metaprogram [CLE95] pode ser comparada com um programa de depuração que monitora o comportamento de uma aplicação, programado para intervir sob determinadas circunstâncias; o código usado para monitoração é separado do código da aplicação, explorando a separação do projeto da aplicação daquilo que a torna tolerante a falhas. A separação não é totalmente transparente visto que é exigida da aplicação a inserção de diretrizes para a intervenção mencionada.

No ambiente Arjuna [SHR95], classes pré-definidas fornecem alguns protocolos e serviços típicos de tolerância a falhas que podem ser usados por objetos da aplicação. Classes parametrizadas são propostas por Rubira [RUB94a] como forma de generalizar estratégias de tolerância a falhas baseadas em objetos redundantes.

No *framework* MOTF é adotada a técnica de reflexão computacional baseada em meta-objetos, como forma de prover as funcionalidades de tolerância a falhas através da interceptação de mensagens a métodos ou objetos críticos e para os quais existe a possibilidade de utilização de redundância *temporal* (ex. repetição de computações), *funcional* (ex. objetos alternativos) ou *espacial* (ex. objetos duplicados).

4.2 O framework MOTF

4.2.1 Arquitetura reflexiva

O projeto de MOTF é um *framework* que apoia o desenvolvimento de aplicações tolerantes a falhas, compreendendo múltiplas classes que definem as funcionalidades exigidas por diversas técnicas de tolerância a falhas. Adota uma arquitetura reflexiva, na qual o meta-nível é dedicado às atividades de detecção e recuperação de erros através da monitoração de objetos da aplicação e gerenciamento de redundância de objetos, localizados no nível base. Características de tolerância a falhas podem ser adicionadas a objetos considerados críticos pela aplicação, assim distribuindo, e não centralizando, a propriedade de tolerar falhas entre objetos da aplicação.

Os requisitos especificados na secção 4.1 são assim atendidos:

a) *Confiabilidade*: os meta-objetos podem ser separadamente testados, reusados e tornados tolerantes a falhas (por exemplo, por replicação).

b) *Separação semântica*: os objetos da aplicação pertencem a classes distintas dos meta-objetos que os controlam.

c) *Modularização*: os mecanismos de tolerância falhas são implementados em classes separadas, que podem sofrer especializações por herança estática.

d) *Composição*: objetos selecionados da aplicação podem ser separadamente associados a meta-objetos, permanecendo inalterados os demais objetos da aplicação.

e) *Especificação de comportamento de alto nível:* classes abstratas definem protocolos dependentes de características particulares do meio ambiente.

4.2.2 Benefícios da arquitetura reflexiva para tolerância a falhas

A organização em níveis, característica essencial de arquiteturas reflexivas, apresenta diversas vantagens: permite o desenvolvimento, teste e reuso independente dos objetos dos dois domínios e distingue os problemas específicos da aplicação dos problemas pertinentes a atividades de tolerância a falhas, como gerenciamento de réplicas e versões. Merecem destaque os seguintes pontos:

- **Separação de classes**: as estratégias de tolerância a falhas adotadas para tornar um *objeto tolerante a falhas* atuam em um meta-nível, de forma que os objetos considerados críticos podem ser instanciados a partir de sua classe original e tem a sua execução controlada em um meta-nível, por objetos instanciados através de outras classes, com capacidade de detectar anomalias na execução de seus referentes e/ou mascarar erros por uso de redundância.

- **Transparência na detecção de erros**: ao interceptar uma mensagem de um cliente a um objeto crítico, o meta-objeto passa a ser o novo requisitante (em lugar do cliente original) do objeto destinatário da mensagem. Em caso de falha do objeto fornecedor do serviço, por exemplo, uma falha sinalizada por mecanismo de exceção, o meta-objeto é capaz de capturar a sinalização da exceção, sem que o cliente original perceba.

- **Preservação de funcionalidade**: outra vantagem desta arquitetura é que as características de tolerância a falhas não interferem na funcionalidade do objeto; o meta-objeto procura apenas assegurar e não modificar a funcionalidade de seu referente. O objeto referente mantém as suas propriedades originais, inclusive a propriedade de ser especificável.

Um outro ponto importante que merece destaque é o reuso dos componentes, abordado a seguir. Tendo em mente a complexidade intrínseca de software tolerante a falhas, a reutilização de componentes

torna-se bastante atraente, pois, além de liberar o programador de aplicação de desenvolver código estranho ao domínio da aplicação, tende a reduzir a incidência de falhas na aplicação.

4.2.3 Reutilização de componentes

É de senso comum que a confiabilidade de software aumenta com a reutilização de código visto que as falhas residuais tendem a decrescer ao longo do tempo de utilização. Por outro lado, o desenvolvimento de aplicações tolerantes a falhas implica implementar, além das funcionalidades específicas da aplicação, mecanismos de tolerância a falhas fortemente acoplados a determinadas porções de código da aplicação, considerados como segmentos críticos.

Vários mecanismos de tolerância a falhas de sistema têm sido implementados de forma reusável, geralmente sob forma de bibliotecas de componentes. Por exemplo, segundo [BEL94], no Departamento de Pesquisa em Engenharia de Software da AT&T foi desenvolvido uma biblioteca de programas reusáveis que implementam pontos de recuperação e um *daemon* para monitoração de processos distribuídos. Esta combinação foi usada com sucesso em mais de vinte aplicações que exigiam tolerância a falhas.

Visando aumentar a possibilidade de reutilização de componentes, duas soluções contribuem igualmente:

a) Definição clara e transparente das interfaces entre objetos tolerantes a falhas e objetos não tolerantes a falhas. Desta maneira, o mecanismo mais adequado para tornar o objeto tolerante a falhas pode ser independentemente desenvolvido, uma vez que a interface permanece estável [ANC95].

b) Possibilidade de reflexão sobre o comportamento de objetos individuais da aplicação, controlando as suas interações com outros objetos sem interferir em seu encapsulamento.

A primeira solução é inerente ao modelo de orientação a objetos, ao definir que a interação entre objetos de uma aplicação é feita única e exclusivamente através de mensagens endereçadas aos seus métodos via interfaces bem definidas e que podem ser transmitidas por herança. Objetos podem herdar propriedades de tolerância a falhas, simplesmente

incorporando novas propriedades e mantendo estáveis as suas interfaces originais, destinadas à interações com outros objetos.

Já a segunda solução pressupõe a separação de atividades e a monitoração das mensagens enviadas ao objeto e induz à utilização de meta-objetos. A utilização de um nível de serviços para atender as funcionalidades da aplicação e de outro nível para (tentar) assegurar que estas funcionalidades sejam atendidas de forma transparente ao usuário dos serviços da aplicação, distingue claramente os dois tipos de atividades e propicia diversas formas de reutilização de componentes, a saber:

- classes e objetos responsáveis por serviços não-críticos da aplicação: estes objetos não sofrem nenhuma alteração pelo fato de participarem de uma aplicação tolerante a falhas;
- classes originais de objetos tolerantes a falhas: para estabelecer a associação entre um objeto da aplicação e um meta-objeto é derivada uma nova classe, denominada *classe reflexiva*. A classe original do objeto continua a existir, podendo ser usada para instanciar objetos não reflexivos.
- classes que definem mecanismos de tolerância a falhas: as classes do domínio de tolerância a falhas são classes comuns de C++, que podem ser reusadas e estendidas.

Os componentes acima mencionados podem ter seu *código* reutilizado. Outra possibilidade é a reutilização do *projeto* de componentes, admitindo concretizações distintas de classes abstratas, mantendo inalterados os seus protocolos.

4.3 Interação de objetos e meta-objetos

Um objeto interage com outros objetos através de suas *interfaces*, endereçadas por mensagens descritas como $O\ op_i\ (args)$, sendo O o nome do objeto destino, op_i a operação solicitada e *args* a lista de argumentos opcionais. A execução de op_i pelo objeto O consiste de uma seqüência de ações, que podem resultar em um ou mais dos seguintes serviços:

- alterar os dados do objeto (estado),
- retornar um ou mais dados como resposta (função),
- criar novos objetos (instância), e,
- enviar mensagens a outros objetos.

O conjunto de *interfaces* de um objeto, descrita como $O\ op_i\ (args)$, para $i=1,2,...,n$, define o *protocolo* do objeto e representa os múltiplos serviços oferecidos pelo objeto. Quando um objeto O é associado a um meta-objeto $\mathcal{M}o$, o protocolo de O é um subconjunto de protocolo de $\mathcal{M}o$, de forma que, quando uma mensagem é enviada a um objeto tolerante a falhas, ela pode ser dirigida para seu meta-objeto de forma transparente ao usuário de seus serviços.

O meta-objeto age como um monitor do objeto base, seu referente: encarrega-se de executar os serviços solicitados pela mensagem, de aceitar, selecionar ou registrar a resposta e de responder ao objeto que solicitou o serviço. Enfim, o meta-objeto pode monitorar seu referente e ampliar os serviços oferecidos, obedecendo ao mesmo protocolo, sem alterar a estrutura dos objetos de nível base.

A título de exemplo, uma aplicação distribuída tem todos os seus processos definidos no nível base, sem preocupar-se com replicação de processos considerados críticos. No meta-nível, meta-objetos são usados para criar objetos replicados, gerenciar a replicação e distribuição dos processos críticos e prover a interface entre os dois níveis.

4.4 Considerações sobre estado de um objeto

O *estado* de um objeto relaciona-se com cada uma das suas ativações. Objetos definem ambientes dinâmicos, visto que são criados e destruídos por mensagens específicas no decorrer da execução do programa.

Além de seu ambiente *local explícito*, que compreende as suas variáveis de instância, há que considerar o seu ambiente *local implícito*, que abrange os dados temporários usados pelo seu ambiente de execução, e seu ambiente de *parâmetros*, que se estabelece a cada mensagem parametrizada recebida. Acrescente-se ainda o seu ambiente *global*, que torna visível ao objeto outros componentes (ex. nomes de variáveis, classes e objetos) não definidos em sua 'cápsula' e sim no seu contexto de definição ou chamada.

A cada ativação de um objeto estas informações são mantidas em um *registro de ativação*, cuja existência se encerra com o término da ativação do objeto. Na semântica de execução de um objeto, o seu registro de ativação reflete o seu estado: seu ambiente local, seus parâmetros e referência a seu ambiente global.

Esse estado, pode-se dizer *estado profundo*, permite ações semânticas como suspender/retomar o fluxo de execução e a sua preservação, por cópia num determinado instante, é conhecida como *ponto de recuperação.*

Nas seções a seguir, o termo *estado de um objeto* possui uma conotação mais superficial. Refere-se somente aos valores de instância e parâmetros com semântica de referência que possam influir numa posterior re-execução do mesmo objeto ou na execução de um objeto alternativo, na prestação de um mesmo serviço a um mesmo cliente.

A possibilidade de efeitos colaterais resultantes da execução parcial de um objeto, a exemplo de alterações feitas em outros objetos, foge do alcance de *frameworks* genéricos de desenvolvimento de aplicações tolerantes a falhas devido à sua forte vinculação ao código do programa.

A utilização de técnicas de tolerância a falhas que utilizam redundância de objetos nem sempre requer a preservação de estado dos objetos participantes. Se o comportamento do objeto é funcional, independente de ativações anteriores e de seu ambiente global, a preservação de estado é desnecessária.

Quando necessária, e dependendo da implementação do objeto, a sua restauração pode ser feita por métodos construtores, lembrando que estes podem ser polimórficos. O uso do polimorfismo pode ser explorado para a implementação de construtores que fazem a inicialização do objeto de forma selecionada de acordo com a circunstância.

Estas ponderações mostram que a confiabilidade pode depender fortemente do projeto da aplicação, e mais especificamente, da implementação interna de cada objeto. Na fase operacional da aplicação, técnicas de tolerância a falhas de software provém alternativas de mascaramento de falhas previstas no projeto da aplicação.

4.5 Objetos reflexivos tolerantes a falhas

Um *objeto tolerante a falhas* é um objeto responsável por algum serviço considerado crítico no contexto da aplicação. Para atender às diferentes exigências de tolerância a falhas, objetos tolerantes a falhas são abstrações de objetos de alta confiabilidade. Portanto, devem ser capazes de fazer uso de diferentes técnicas para assegurar o contínuo e (presumivelmente) correto fornecimento de seus serviços.

Em uma aplicação, objetos tolerantes a falhas e objetos não tolerantes a falhas podem coexistir, e clientes que solicitam os serviços de um objeto não estão cientes da condição crítica do serviço solicitado. A diferença essencial entre um objeto tolerante a falhas e outro objeto qualquer da aplicação é que o primeiro possui um meta-objeto a ele associado. Os objetos que compõem a aplicação interagem entre si, atendendo aos serviços da aplicação e aqueles considerados críticos são controlados pelos meta-objetos. Os objetos tolerantes a falhas podem ser dinamicamente criados pelos meta-objetos no decorrer da execução da aplicação, de acordo com a técnica de tolerância a falhas de que participam.

Os meta-objetos pertencem a meta-classes que, além da propriedade de reflexão computacional, definem os mecanismos e serviços típicos de cada técnica. Visto que o comportamento tolerante a falhas de um objeto da aplicação é determinado pelo meta-objeto a ele associado, é possível que objetos instanciados a partir da mesma classe sejam sujeitos a diferentes estratégias de tolerância a falhas, como ilustrado na figura 4.1. As meta-classes correspondem às diversas classes definidas pelos MOTFs- Serviços, porém possuem a propriedade de reflexão computacional fornecida através de um MOP- Meta-Object Protocol[KIC91].

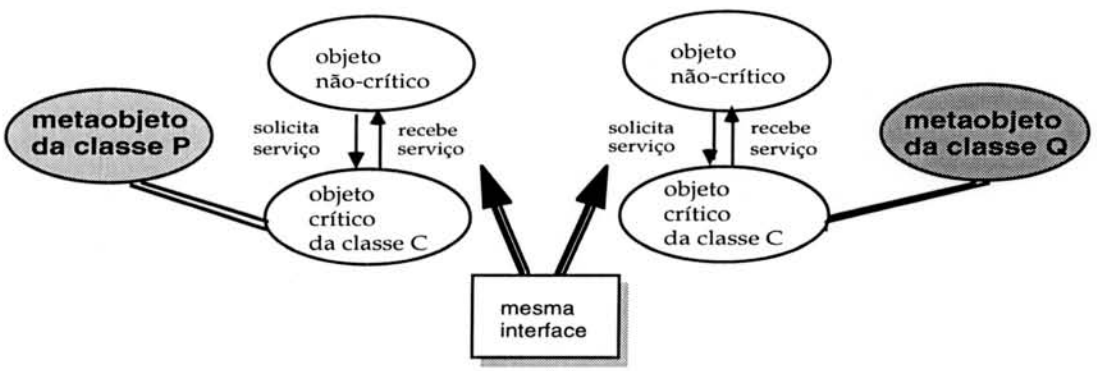


Figura 4.1 - Objetos reflexivos tolerantes a falhas

A arquitetura do MOTF corresponde ao modelo adotado em OpenC++[1], uma extensão reflexiva de C++, desenvolvida por Chiba [CHI93b]. Esta extensão baseia-se em um pré-processador que permite associar de

[1] Ver Anexo 2.

forma estática meta-objetos a objetos da aplicação. No nível da aplicação, o programador pode escolher quais objetos devem ser reflexivos e, em cada objeto, quais métodos e quais atributos devem ser controlados pelo meta-objeto a ele associado.

No meta-nível, para onde são 'desviadas' todas as chamadas feitas a métodos reflexivos, o programador especifica a computação a ser feita para fins de controle, monitorização, complementação ou modificação da computação original do nível base. Desta forma, são definidas diversas meta-classes instanciáveis em meta-objetos distintos para a implementação de blocos de recuperação, programação em n-versões e outros para controle de grupos de objetos.

Um meta-objeto é considerado como um objeto da aplicação, admitindo ser controlado por um outro meta-objeto, formando assim a infra-estrutura para a combinação de técnicas de tolerância a falhas e para a utilização de serviços oferecidos pela máquina virtual da aplicação tolerante a falhas.

O gerenciamento da distribuição, pré-requisito para a implementação de protocolos de replicação de objetos, usa serviços que são oferecidos a nível de sistema, como por exemplo, serviços de difusão de mensagens. Portanto, mais um nível de computação é introduzido, ficando a torre de reflexão[2] assim organizada [ANC95]:

- Nível D_0: contém os objetos da aplicação, que podem ser críticos ou não críticos, sendo os primeiros controlados por meta-objetos.
- Nível D_1: composto pelos meta-objetos responsáveis pelas atividades de tolerância a falhas.
- Nível D_2: contém as primitivas de suporte às ações de tolerância a falhas, a exemplo de primitivas de organização de grupos distribuídos e de difusão de mensagens em sistemas distribuídos.

O meta-meta-nível D_2 atua como interface dos níveis D_0 e D_1 com outros sistemas que suportam computação distribuída e serviços de tolerância a falhas de sistemas. Este nível adicional permite, por exemplo, a replicação de meta-objetos com vistas à tolerância a falhas de sistema.

---

[2] Ver Cap. 3, seção 3.5.

4.6 Grupos de objetos

Uma característica importante do modelo de orientação a objetos e relevante para o desenvolvimento de aplicações tolerantes a falhas é o *encapsulamento*. Se dois objetos o_i e o_k possuem o mesmo protocolo e o mesmo comportamento externo, estes objetos podem ser considerados *equivalentes*, independentemente de sua estrutura interna. A substituição de uma instância qualquer de o_i por o_k não causará problemas à aplicação, desde que o mesmo serviço seja realizado. Esta equivalência comportamental admite um gerenciamento unificado de réplicas e versões de objetos: dado um conjunto de objetos equivalentes $o_1, o_2,...,o_n$ que implementam a mesma semântica, cada objeto o_i pode ser indistintamente uma versão ou uma réplica de o_k.

Esta abstração permite que um cliente interaja com múltiplos objetos através de um único protocolo. Os comportamentos de objetos tolerantes a falhas acima mencionados pressupõe a existência de redundância, seja por diversidade ou por replicação de componentes, bem como o fornecimento de algum serviço: um objeto recebe uma solicitação de serviço e dele é esperada a execução deste serviço. Portanto, faz sentido organizar a redundância através de *grupos de objetos*, cujo comportamento externo aparente os serviços de um único objeto.

Neste sentido, um *grupo de objetos* possibilita que falhas parciais sejam mascaradas e que membros do grupo sejam modificados sem que seus clientes percebam. Todos os membros do grupo obedecem à mesma estratégia de tolerância a falhas, possuem o mesmo tipo e podem ser réplicas de um mesmo objeto ou objetos diversos de igual protocolo.

O conceito de grupo serve de base à implementação de diversas técnicas, como programação em n-versões, blocos de recuperação e replicação de objetos. As duas primeiras técnicas podem ser aplicadas em ambientes centralizados. A técnica de replicação de objetos é adequada à adoção de tolerância a falhas em ambientes distribuídos; neste caso, grupos de objetos podem ser replicados em nodos distintos, usando um segundo nível de reflexão computacional, D_2 . O nível D_1, na forma de um objeto reflexivo ou de um grupo de objetos, passa a ser a unidade de distribuição e replicação.

No caso de objetos diversificados, o serviço pode ser oferecido pelos meta-objetos de duas formas:

- *Individual*: o objeto *O* é controlado pelo meta-objeto *Mo* e, em caso de falha de *O*, a computação passa a ser feita pelo objeto *O'*, funcionalmente equivalente a *O*.
- *Grupo*: o meta-objeto *Mo* controla um grupo de objetos *O*, *O'*, *O"*, ..., funcionalmente equivalentes. A solicitação de um serviço crítico é atendida por *Mo* , que se encarrega de ativar os membros do grupo, selecionar ou aceitar o resultado da computação e responder ao solicitante (figura 4.2).

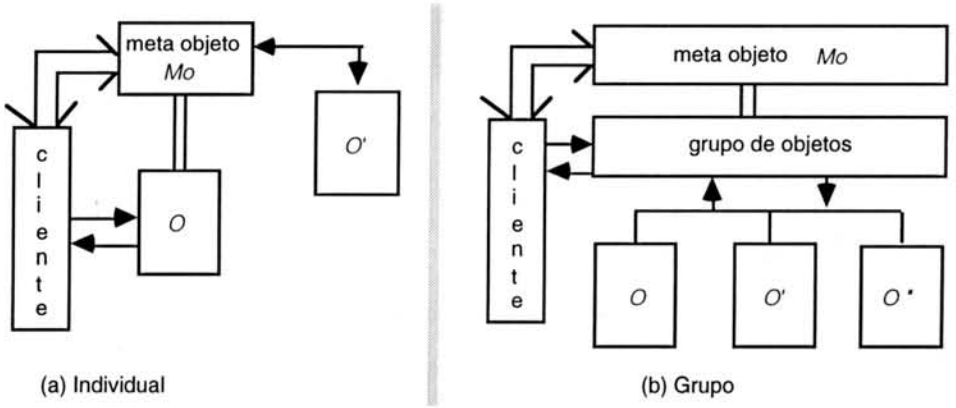


Figura 4.2 - Objetos diversificados

Neste nível de granularidade, pode-se obter simultaneamente a tolerância a falhas no sistema e na aplicação, por replicação de grupos de objetos. Como cada grupo de objetos é controlado por um meta-objeto (nível D_1), a replicação de grupos implica em uma replicação de meta-objetos, os quais passam a ser controlados por um meta-meta-objeto (nível D_2).

4.7 A semântica da redundância

Um objeto tolerante a falhas representa um grupo de objetos redundantes, controlados por um meta-objeto que determina a semântica de interação entre os componentes redundantes e seu controlador: votação ou 'primeira-resposta'.

A *semântica de votação* requer um grupo redundante de pelo menos três componentes, usados para atender de forma concorrente o mesmo

serviço. Suas múltiplas respostas são submetidas a uma votação para a seleção de uma resposta a ser devolvida ao cliente.

A *semântica de 'primeira-resposta'* baseia-se no recebimento de apenas uma resposta de um membro do grupo e não requer que todos os membros sejam ativados concorrentemente. Esta resposta pode ser submetida a um julgamento antes de ser devolvida ao cliente.

Essas semânticas podem ser implementadas por replicação ou por diversidade de componentes, dependendo do objetivo da redundância e da configuração do sistema de suporte disponível (monoprocessamento, multiprocessamento ou processamento distribuído).

A semântica de votação subdivide-se em duas categorias:

- Componentes diversificados: utiliza componentes com projeto/implementação distintos, denominados de versões ou variantes, e corresponde à técnica clássica de programação em n-versões. Pode ser implementada em ambientes centralizados, admitindo concorrência para minimização do tempo total de processamento.
- Componentes replicados: utiliza componentes com estrutura e estado inicial idênticos executados concorrentemente e corresponde à técnica conhecida como replicação ativa. Exige suporte de programação paralela ou distribuída.

Da mesma forma, a semântica de primeira-resposta admite as variações:

- Componentes diversificados: utiliza componentes com projeto/implementação distintos, denominados de alternativas, e corresponde à técnica clássica de blocos de recuperação. Idealizada para ser implementada em ambientes centralizados, admitindo concorrência para reduzir o tempo de processamento em caso de falha da alternativa primária.
- Componentes replicados: utiliza componentes com estrutura e estado inicial idênticos, exigindo que apenas um esteja ativo e os demais podem ou não ser executados concorrentemente. Corresponde às técnicas conhecidas como replicação passiva,

'espera-a-quente' ou 'espera-a-frio'. Exige suporte de programação paralela ou distribuída.

Cada uma dessas semânticas admite diferentes abordagens de técnicas de implementação ou *esquemas* de tolerância a falhas. Xu [XU94] classifica os esquemas de tolerância a falhas com componentes diversificados em duas categorias: redundância estática (representada pela programação em n-versões) e redundância dinâmica (representada por blocos de recuperação).

A classificação semântica aqui proposta diverge da classificação de Xu por considerar a semântica de votação como *redundância ativa* (incluindo programação em n-versões) e a semântica de primeira-resposta como *redundância passiva* (incluindo blocos de recuperação). Esta classificação corresponde à semântica de execução dos objetos redundantes.

4.8 Granularidades de tolerância a falhas

Para a composição de uma aplicação, os objetos tolerantes a falhas podem se apresentar em diferentes graus de diversidade e/ou replicação, como identificadas em [CLE95]:

- *Diversidade de métodos*: pelo menos um método de um objeto é diversificado. O método pode acessar o estado do objeto, porém não pode alterá-lo, evitando assim efeitos colaterais da computação do método.
- *Diversidade de objetos:* diversos objetos implementam a mesma funcionalidade, definem a mesma interface porém diferenciam-se pela sua implementação concreta, podendo ter diferentes estruturas e, em decorrência disto, não possuem um estado comum.
- *Replicação de objetos*: múltiplas instâncias de uma mesma classe são criadas, gerando objetos inicialmente idênticos com o objetivo de tolerar falhas no sistema. As réplicas podem ser distribuídas, possibilitando a existência de estados diferentes entre as réplicas no decorrer da computação.

As diferentes granularidades de tolerância a falhas adotam diferentes estratégias de reflexão computacional: no caso mais simples, cada objeto crítico é controlado por um meta-objeto. Em outros casos, quando mais de

um objeto participa da ação de tolerância a falhas, um meta-objeto controla um grupo de objetos sujeitos à mesma estratégia de tolerância a falhas e cada objeto do grupo pode ser controlado por um meta-objeto, com a finalidade de salvamento/restauração de estado ou de execução distribuída de seu referente.

4.8.1 Diversidade de métodos

A diversidade de métodos dentro de um mesmo objeto é o caso mais simples, visando a tolerância a falhas em software. A idéia é que exista pelo menos um método crítico em um objeto, e que mais de uma implementação é utilizada para aumentar a confiabilidade desse método. O método pode acessar o estado do objeto, mas o estado não é alterado (comportamento funcional), preservando a integridade do objeto após a execução do método. A distribuição de objetos não é um pré-requisito para este nível de granularidade.

A diversificação de métodos de um objeto é possível sempre que existir mais de um algoritmo para implementar a funcionalidade desejada. Para adotar a técnica de programação em n-versões, são necessárias pelo menos três versões para possibilitar a votação de um resultado por maioria. A técnica de blocos de recuperação pode ser implementada com apenas duas alternativas, uma primária e uma secundária, mas exige a existência de uma função de aceitação que permita determinar se o resultado pode ser aceito como correto ou não. Caso a tolerância a falhas por *valor* (resultado correto) possa ser desprezada em função da tolerância a falhas por *omissão* (sempre que for fornecido um resultado, este é aceito) ou por *tempo* (o resultado deve ser fornecido dentro de um período de tempo pré-determinado), a função de aceitação pode ser substituída por um sistema de tratamento de exceções ou por uma função de controle de tempo.

A diversidade de métodos admite as seguintes variações:

- Objeto único que contém as diferentes alternativas do método crítico, ou,
- Classe única, com diversas instâncias contendo as diferentes implementações do método crítico.

O primeiro caso, adequado à técnica de blocos de recuperação, é de fácil implementação, dispensando um gerenciamento de redundância mais

complexo. No modelo de programação imperativa, esta é a abordagem clássica de *código crítico*, implementado sob forma de funções pertencentes a um mesmo ambiente global.

No segundo caso pode ser aplicada a técnica de programação em n-versões. sendo cada instância uma unidade de execução, dispensando mecanismos de concorrência *interna* de objetos para a sua execução. Também pode ser usada para combinação de técnicas: cada objeto possui métodos alternativos, com ordem de prioridade distinta (o primário de uma instância corresponde ao secundário de outra e assim por diante) e as diversas instâncias - cada uma representando uma versão - são executadas concorrentemente e seus resultados submetidos a duplo julgamento. Um julgamento interno, por teste de aceitação, e um julgamento externo, por votação. Abordagem semelhante é sugerida por Laprie [LAP90], em processos que combinam as técnicas de programação em n-versões e blocos de recuperação, com base em uma técnica originada de tolerância a falhas em hardware, a redundância ativa dinâmica.

Para o objeto usuário dos serviços do objeto tolerante a falhas passa desapercebida a existência de diversos métodos que implementam a mesma funcionalidade, mas o mesmo não acontece ao projetista da aplicação. Para a utilização de métodos diversificados em um mesmo objeto, a aplicação se preocupa em fornecer as diversas implementações dos métodos e os métodos de votação ou aceitação.

4.8.2 Diversidade de objetos

Neste caso, todo o objeto é considerado crítico, e não apenas um de seus métodos. A mesma abstração possui diferentes concretizações. Por exemplo, a abstração de uma pilha pode ser implementada usando estruturas de dados distintas. Ou ainda, o objeto é considerado crítico porque seu estado é alterado em cada ativação. A implementação das funções alternativas como objetos distintos, facilita a preservação das variáveis de estado (os dados do objeto), para uma eventual ação de retrocesso do estado da computação e possibilita o confinamento dos danos causados por ocorrências de erros. Aqui podemos distinguir dois tipos de diversidade:

- *Mesma estrutura, diferentes métodos*: os diversos objetos descendem de uma única classe, que contém uma ou mais funções virtuais que são implementadas de forma distinta. O

estado do objeto pode ser preservado em uma única cópia, os diferentes objetos/métodos são executados, seleciona-se por votação ou aceitação um único resultado e decide-se por um único estado correto de forma a manter a consistência dos diversos objetos

- *Diferentes estruturas e métodos*: os objetos pertencem a classes distintas e não existe um estado comum. Os objetos são executados e a votação ou aceitação é feita com base nos seus resultados. Para que seja feita uma reconstrução de estado em algum objeto falho é necessário conhecer os detalhes das implementações.

A diversidade de objetos para fins de tolerância a falhas é o ponto forte deste modelo de programação, principalmente quando é obtida por redefinição de métodos, possibilitando a cópia de objetos da mesma forma simplificada que os demais modelos de programação permitem copiar dados de tipos primitivos - por comando de atribuição.

Para obtenção de diversidade de objetos, usados como alternativas ou versões, conceitos típicos deste modelo revelam e justificam a sua importância: definição de classes (simples, abstratas, parametrizadas), herança (genérica ou seletiva), polimorfismo e criação de objetos por instanciação.

4.8.3 Replicação

Quando o objetivo é tolerar falhas de sistema, as técnicas de diversificação cedem lugar às técnicas de replicação, mais adequadas sob o ponto de vista de custo de desenvolvimento. Como mencionado anteriormente, a replicação de objetos pode ser obtida por instâncias de uma mesma classe. A distribuição destas réplicas em nodos distintos de uma rede possibilita a existência de estados diferentes entre as réplicas no decorrer da computação, exigindo a adoção de um protocolo de replicação.

Protocolos de replicação buscam assegurar, além de um estado consistente entre as réplicas, a disponibilidade de um número mínimo de componentes, em estado ativo ou passivo. Na arquitetura reflexiva, protocolos de distribuição e replicação são implementados no meta-nível [CLE95], [FAB95], [CHI93b]. Aqui, novamente, a vantagem se concentra na

separação dos objetos da aplicação dos objetos que gerenciam a distribuição e replicação.

4.9 Aspectos de implementação

O MOTF deve participar do processo de desenvolvimento da aplicação tolerante a falhas. Portanto, este *framework* de tolerância a falhas consiste de ferramentas para a configuração da aplicação e diversos MOTFs-Serviços, representando classes de serviços.

Uma aplicação pode ser vista como uma composição de três tipos de objetos: os objetos da aplicação que devem participar de alguma técnica de tolerância a falhas, os objetos que fornecem serviços de tolerância a falhas e os demais objetos da aplicação.

Para configurar a aplicação tolerante a falhas é necessário: (a) identificar os objetos tolerantes a falhas da aplicação, determinando as técnicas a serem usadas; (b) incluir as meta-classes correspondentes e os componentes redundantes e os demais exigidos (por exemplo, para testes de aceitação), de acordo com a técnica selecionada.

Nesta seção são apresentadas algumas soluções de implementação e discutidas algumas particularidades das técnicas de programação que influem na implementação. Exemplos de aplicação destas técnicas são esquematizados para mostrar as diferentes granularidades, com ênfase na separação de funcionalidades.

4.9.1 Criação de um grupo de objetos

Um grupo de objetos é sempre criado de acordo com os requisitos de uma técnica de tolerância a falhas, visto que:

- todos os membros do grupo obedecem à mesma técnica de tolerância a falhas;
- um cliente de um grupo interage com apenas um objeto tolerante a falhas, o qual esconde do cliente os múltiplos objetos (diversificados ou replicados) que ele representa.

Uma vez selecionada a técnica de tolerância a falhas, a criação dos membros do grupo é idêntica para todas as técnicas, sob a responsabilidade do meta-objeto associado ao objeto tolerante a falhas, lembrando sempre que

é incumbência da aplicação prover a definição dos objetos participantes do grupo.

Um grupo de objetos é representado na aplicação por um objeto reflexivo tolerante a falhas, a seguir descrito. Em outras palavras, um cliente dirige a sua mensagem solicitando serviços a um objeto tolerante a falhas; nos 'bastidores' desta chamada, existe uma estrutura de dados que contém todos os possíveis objetos que podem fornecer o serviço solicitado e que são ativados pelo meta-objeto associado ao objeto tolerante a falhas.

4.9.2 Criação de um objeto reflexivo tolerante a falhas

Um objeto reflexivo tolerante a falhas atua como uma interface a um grupo de objetos com as seguintes características:

- é um objeto reflexivo, tendo a ele associado um meta-objeto da classe correspondente à técnica de tolerância a falhas a ser aplicada;
- possui pelo menos um método reflexivo, que determina a interface comum a todos os objetos que ele representa;
- seu comportamento (por exemplo, execução alternada ou simultânea) é determinado pela técnica de tolerância a falhas.

A criação de um objeto reflexivo tolerante a falhas é feita com base nas seguinte informações fornecidas pela aplicação: classe MOTF correspondente à técnica, definição das classes dos objetos participantes e definição do objeto interface, com indicação do método reflexivo.

O objeto interface pode ser interpretado de duas maneiras, por escolha do implementador: pode ser uma instância de uma classe concreta, implementando um método crítico que possui alternativas controladas pelo meta-objeto ou pode ser uma instância de uma classe abstrata, que não concretiza a implementação, servindo apenas como interface para as implementações concretas, todas elas controladas no meta-nível.

A seguir são mostrados alguns exemplos de criação de objetos reflexivos tolerantes a falhas, implementando diferentes estratégias.

4.9.3 Implementação de blocos de recuperação

Testes de aceitação ancoram a técnica de blocos de recuperação, que utilizam computações alternativas para funções críticas do sistema. A implementação desta técnica pode exigir que, quando uma alternativa for executada no lugar da função original, o estado do sistema seja restaurado. No modelo de orientação a objetos, a técnica de blocos de recuperação pode ganhar novos contornos e novas dificuldades, quando consideradas as diversas granularidades de componentes críticos.

Na técnica de blocos de recuperação, a granularidade torna-se importante pois deve ser feita a restauração do estado da computação antes da execução da próxima alternativa. Na arquitetura MOTF, a restauração é de responsabilidade do meta-objeto, de acordo com a granularidade, a saber:

Método crítico: considerando que os métodos alternativos acessam idênticos dados de instância, devem ser definidos como reflexivos os dados que podem ser alterados na execução do método crítico. O meta-objeto encarrega-se de preservar uma cópia dos dados reflexivos antes da execução do método crítico e de fazer a restauração de estado antes da execução de cada alternativa.

Objeto crítico, com alternativas descendentes de uma mesma classe: considerando que as classes descendentes, além dos dados comuns recebidos por herança podem ter dados próprios, são preservados pelo meta-objeto, através de cópia, apenas os dados comuns, para fins de restauração de estado; os dados próprios a serem preservados/restaurados como parte do estado do objeto (estado histórico), devem ser especificados por redefinição das funções de cópia e restauração.

Objeto crítico, com alternativas descendentes de classes distintas: neste caso não existe um estado comum entre as alternativas. O usuário deve redefinir as funções de cópia e restauração a serem usadas pelo meta-objeto.

4.9.4 Exemplo de métodos alternativos

Supondo a abstração de uma classe responsável por serviços de interpolação polinomial. Concretamente, objetos desta classe recebem uma série de valores, que representam as coordenadas de pontos e calculam as coordenadas de um ponto intermediário por interpolação, no caso, pelo

método linear. Uma classe com esta funcionalidade pode ser assim esquematizada:

```
class Interpola{
public:
double InterpolaValor (.....)        /*  calcula o valor interpolado */
private:
double Erro (.....)          /* calcula o erro de interpolação */;
};
```

Figura 4.3 - Esquema da classe Interpola

O método InterpolaValor, que implementa a interpolação linear, é bastante rápido mas apresenta resultados imprecisos quando é grande a distância entre os pontos do intervalo de interpolação. Uma possível solução é prover duas diferentes implementações, uma primária e uma alternativa, e um teste de aceitação baseado no erro calculado. Um meta-objeto é usado para controlar a execução das alternativas, implementando os serviços de blocos de recuperação e registrando algumas informações a respeito dos métodos controlados.

No exemplo em pauta, a função alternativa é implementada em outra classe, como uma interpolação de Lagrange, e o teste de aceitação é definido no meta-objeto. Esta estratégia foi adotada para a implementação de blocos de recuperação sem modificar a classe original (que pode ainda ser instanciada como outros objetos não tolerantes a falhas) e separando completamente o código da aplicação do código de tolerância a falhas, como a seguir esquematizado na figura 4.4.

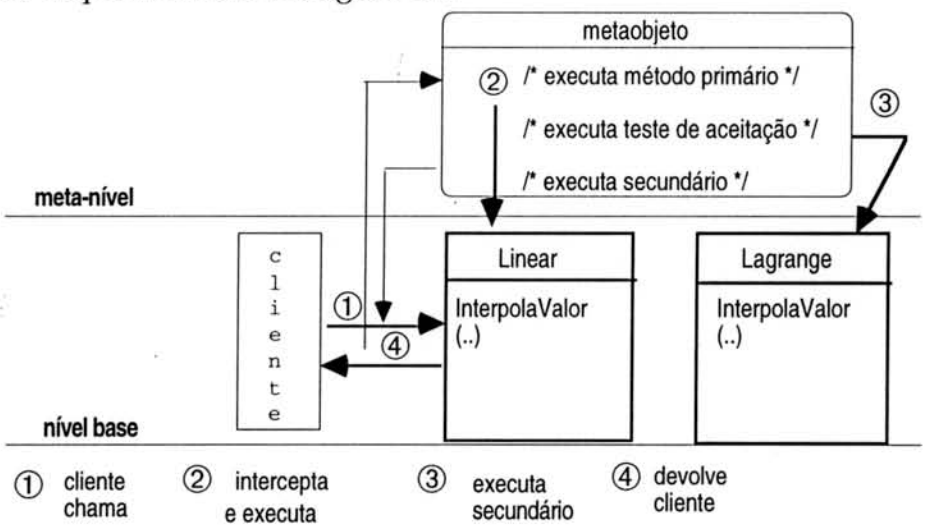


Figura 4.4 - Meta-objeto controlando um método

Este esquema apresenta ainda como o meta-objeto intercepta a chamada ao método crítico. A mensagem originada pelo cliente

inter.InterpolaValor (X, eixoX,eixoY,error) :

① é interceptada pelo meta-objeto, que realiza as seguintes computações a seguir.

② chama o método de interpolação linear (InterpolaValor) e submete seu resultado ao teste de aceitação.

③ em caso de não aceitação do resultado, o meta-objeto realiza sem interferência e conhecimento do cliente, o cálculo da interpolação por Lagrange com três pares de pontos, que produz um erro menor porém com custo computacional maior.

④ um resultado é devolvido ao cliente.

Uma outra abordagem em relação à determinação do método primário também pode ser adotada no meta-nível. O meta-objeto pode coletar dados estatísticos a respeito da execução do método primário (neste exemplo, o de interpolação linear) e, com base nestes dados, executar como método primário o originalmente classificado como alternativo, invertendo a ordem de chamada. Esta abordagem explora uma das características da técnica de reflexão computacional: a de 'raciocinar' sobre as computações de nível base, tirar conclusões e traduzir estas conclusões em ações no nível base.

Para a realização da reflexão computacional, o método crítico do objeto deve ser identificado: a figura 4.5, usando a sintaxe de OpenC++, mostra a classe Interpola indicando como reflexivo o método InterpolaValor.

```
class Interpola{
public:
//MOP reflect:
double InterpolaValor (.....)        /* calcula o valor interpolado */
private:
double Erro (.....)          /* calcula o erro de interpolação */;
};
```

Figura 4.5 - Definição da classe de nível base com método reflexivo

O meta-objeto da classe MotfRB, que define o protocolo do meta-objeto e os serviços de blocos de recuperação, é então associado à classe Interpola, usando uma diretiva para o pré-processador de OpenC++ (figura 4.6).

```
// MOP reflect class Interpola: MotfRB
```

Figura 4.6 - Associação ao meta-objeto da classe MotfRB

O pré-processador redefine a classe Interpola, incluindo a mensagem para o meta-objeto (Meta_MethodCall). Quando o método crítico InterpolaValor recebe uma mensagem, esta é refletida em seu meta-objeto, contendo informações sobre seu nome e seus argumentos.

Na interceptação de uma chamada a um objeto reflexivo, os parâmetros sofrem um processo de *reificação*, para serem tornados disponíveis ao meta-objeto. Em OpenC++, a reificação é restrita a tipos primitivos de dados, sendo implementada em um objeto denominado ArgPac, que utiliza uma estrutura de pilha. O meta-objeto utiliza a pilha de argumentos para chamar o seu referente (Meta_HandleMethodCall), para executar outros objetos com os mesmos argumentos, mantendo cópia dos argumentos originais de chamada.

Se o resultado da versão primária não for aceito, a alternativa que calcula a interpolação por Lagrange é executada pelo meta-objeto, de forma análoga. A figura 4.7 mostra uma implementação simplificada do meta-objeto.

```
void MotfB::Meta_MethodCall(int m_id, int cat,
                            ArgPac& args, ArgPac rep)
{.....
   Meta_HandleMethodCall (m_id,args,rep); /* chama o método primário*/
   if (ACC(*erro))        {                        /* teste de aceitação */
         Lagrange lag ;          /* instancia objeto da classe Lagrange */
         lag.InterpolaValor(....) ; /* interpola por Lagrange */
   }
}
```

Figura 4.7 - Definição do metaobjeto

Esta técnica de componentes alternativos naturalmente onera a aplicação, em custo de desenvolvimento e processamento. Porém, além de tolerar a falhas, blocos de recuperação também podem ser usados para

suavizar a transição de um componente do sistema para uma nova versão, mantendo a confiabilidade do sistema, como nesta antiga sugestão de Hecht [HEC76]:

> "Uma aplicação de blocos de recuperação particularmente atraente é onde um módulo existente porém ineficiente está sendo substituído por outro de desempenho superior porém de confiabilidade ainda não comprovada. O módulo existente pode então ser usado como alternativa, diminuindo os custos do software."

Adaptando esta sugestão ao modelo de orientação a objetos, um módulo pode corresponder a um método, a um objeto ou mesmo a um grupo de objetos de menor granularidade, que se associam para produzir um serviço, e esta transição pode ser controlada por um meta-objeto, semelhante ao descrito nesta seção.

4.9.5 Implementação de programação em n-versões

A implementação desta técnica requer dois cuidados especiais: o gerenciamento da execução das versões e a votação dos resultados.

Gerenciamento da execução

É feito usando um único meta-objeto e um grupo de objetos de nível base. Um deles é o *objeto tolerante a falhas* usado apenas como interface de recebimento de mensagens e os outros são as versões (operacionais) V_i, com $i=1,2,...,n$. O meta-objeto é responsável pela execução das versões, coleta de resultados, controle de votação e registro dos resultados obtidos da computação das versões. A computação se desenvolve da seguinte maneira:

a) O objeto tolerante a falhas recebe uma solicitação de execução de um serviço crítico. Esta mensagem é capturada pelo meta-objeto controlador do grupo, que reenvia esta mensagem a todas as versões que fornecem este mesmo serviço crítico e aguarda os resultados.

b) Cada versão, em caso de execução distribuída é controlada por um meta-objeto: o objeto de nível base executa a versão e o resultado da execução é devolvido ao meta-objeto.

c) Quando os resultados da execução das n-versões são recebidos, meta-objeto promove a votação e devolve o resultado do consenso ao cliente que solicitou o serviço.

A utilização da técnica de programação em n-versões [AVI85] exigirá da aplicação as seguintes especificações:

a) Objetos participantes: Definição dos objetos ou métodos participantes da programação diversitária. Todos os participantes implementam o mesma funcionalidade e seu código não sofre qualquer alteração pelo fato de participar da programação diversitária.

b) Interfaces dos objetos participantes: Todos os objetos participantes deverão possuir a mesma interface, visto que os dados de entrada serão idênticos para todas as versões e os dados resultantes da execução deverão ser de mesmo tipo.

c) Algoritmo de votação: A aplicação poderá utilizar um dos algoritmos de votação fornecidos pela classe Votador ou poderá fornecer o seu próprio algoritmo de votação.

d) Restrições: A aplicação poderá indicar restrições de tempo de execução dos objetos participantes, para controle de sincronização das versões.

e) Versões faltosas: A aplicação poderá fornecer diretivas particulares para o tratamento de versões faltosas.

A votação

Em um *framework* genérico de suporte à programação em n-versões, a votação pode oferecer problemas quando os resultados são representados em ponto flutuante ou quando existe a possibilidade de múltiplas respostas corretas porém distintas [JAL94][KEL91]. Este último caso é fortemente dependente do domínio da aplicação e requer solução específica.

Sobre um conjunto de valores similares (raramente idênticos), a votação *imprecisa* deve ser empregada para selecionar um valor final com base em critérios como tolerância de erros, valor médio ou mediano [JAL94].

O modelo de orientação a objetos oferece um bom suporte para a implementação de algoritmos de votação genéricos. Classes parametrizadas permitem variações nas estruturas de dados usadas no armazenamento de resultados e o polimorfismo admite variações nos critérios de seleção de um resultado final, a exemplo de maioria absoluta, maioria relativa, média ou mediana de maioria, entre outros.

Os resultados das versões são armazenados em um vetor de resultados [AVI85], [KEL91], representado no *framework* MOTF como uma abstração de um *Array* de N elementos do tipo T, esquematizado por uma classe parametrizada, na figura 4.8:

```
template <class T>
class Array  {
public:
        Array (int N); /*construtor */
}
```

Figura 4.8 - Classe genérica do vetor de votação

Os métodos polimórficos de votação são reunidos em uma classe Votador, cujas instâncias podem ser tornadas tolerantes a falhas segundo a mesma técnica usada nos objetos tolerantes a falhas. Um meta-objeto é associado ao objeto da classe Votador, encarregando-se do gerenciamento dos diversos métodos de votação ou da re-execução de um mesmo método, com ligeiras variações nos argumentos de chamada (ex. tolerância de erro).

Exemplo: Supor o conjunto de 10 valores em ponto flutuante:

2.7178	2.7182	2.7150	2.7181	2.7182	2.7182	2.7184	2.7183	2.7182	2.7183
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)

Sobre este conjunto podem ser aplicados diferentes critérios de seleção de um resultado, obtendo-se variação nas respostas:

Critério	**Resposta**	**Nº participantes**
Maioria absoluta	Sem resposta	0
Maioria absoluta, 4 dígitos	2.718	8
Média dos valores	2.7178	10

Se o meta-objeto controlador seguir a estratégia de n-versões para selecionar um resultado final dentre os resultados fornecidos pelos diferentes votadores, o método de seleção passa a ser recursivo.

4.9.6 Exemplo de objetos diversitários

O exemplo a seguir adapta o exemplo definido para a técnica de blocos de recuperação e supõe que os objetos diversificados descendem de uma mesma classe, denominada Interpola. As classes descendentes

implementam a interpolação polinomial usando três diferentes métodos. Os diversos objetos são controlados por um único meta-objeto, da classe MotfNVP, que fornece os serviços de programação em n-versões. O cliente interage com o objeto da classe Interpola, sem tomar conhecimento que este objeto possui três versões.

Na execução, a chamada para InterpolaValor é refletida para o meta-nível, onde a interpolação é calculada por três diferentes algoritmos: Linear, Lagrange e Newton. O valor resultante da interpolação retornado ao cliente é obtido por votação por um método implementado pelo meta-objeto. A figura 4.9 apresenta a estrutura de execução das três versões, controladas por um único meta-objeto. Nesta representação, as setas identificadas por ③ identificam as chamadas realizadas pelo meta-objeto aos objetos interpoladores definidos no nível base.

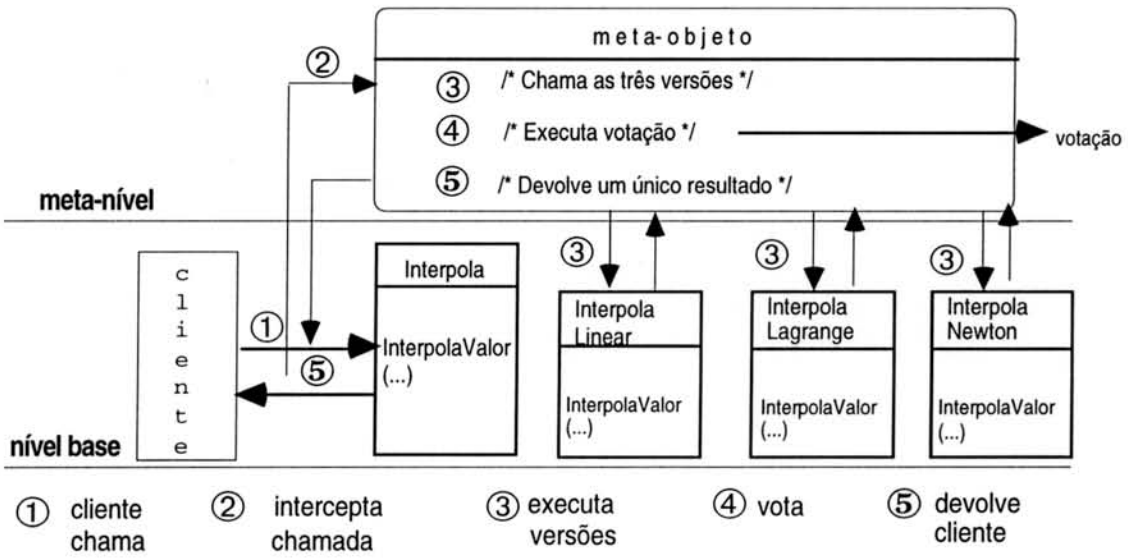


Figura 4.9 - Meta-objeto controlando objetos diversificados

Nos 'bastidores' deste exemplo, os diversos objetos são agrupados em uma interface comum, suportada por um único meta-objeto. Neste caso, a classe do nível base também é chamada de Interpola, possuindo a mesma definição da classe base com método reflexivo que utiliza a técnica de blocos de recuperação anteriormente esquematizada na figura 4.3. A diferença é que esta classe, no caso de diversificação de objetos não possui uma implementação concreta para os métodos InterpolaValor e Erro.

A atividade de cálculo é realizada pelas três versões, implementadas pelas classes descendentes InterpolaLinear, InterpolaLagrange e InterpolaNewton. As classes descendentes concretizam os métodos abstratos da classe original do objeto reflexivo tolerante a falhas.

A diferença de comportamento do objeto da classe Interpola nos exemplos de blocos de recuperação e programação em n-versões foi ditada pelo meta-objeto correspondente a cada técnica. A partir de uma única classe, foram feitas as associações:

Bloco de recuperação: // MOP reflect class Interpola: MotfRB

Programação em n-versões: // MOP reflect class Interpola: MotfNVP

A mensagem enviada pelo cliente ao objeto reflexivo tolerante a falhas e a instanciação deste último permaneceu inalterada:

Instanciação: refl_Interpola inter;

Mensagem: Y= inter.InterpolaValor(X, eixoX,eixoY,error)

É oportuno lembrar aqui que a figura 4.1 esquematiza esta diferença comportamental no meta-nível, mantidas inalteradas as interfaces no nível base.

4.10 Conclusões

É sabido que o custo de desenvolvimento de software tolerante a falhas é muito alto, inviabilizando a sua plena utilização e tornando estas técnicas de utilidade questionável. Em contrapartida, a única forma conhecida de tolerar falhas de projeto e programação, na fase operacional do software, é a diversidade de projetos e de implementação. No caso de replicação de componentes, o custo associado refere-se ao gerenciamento das réplicas.

Uma possível solução para este problema consiste em reduzir os custos de desenvolvimento de software tolerante a falhas, de duas formas:

i- Simplificar o processo de desenvolvimento de software tolerante a falhas e,

ii- Aumentar a possibilidade de reuso de componentes.

O *framework* MOTF - Meta-Objetos para Tolerância a Falhas, aqui apresentado se propõe a realizar estes objetivos através de:

a) Adoção do modelo de orientação a objetos, explorando a possibilidade de instanciação múltipla, a programação 'por diferença', o encapsulamento do estado do objeto e a distribuição de características de tolerância a falhas a objetos selecionados da aplicação;

b) Adoção de uma arquitetura reflexiva, que utiliza separação de classes e de domínios, promovendo a reutilização dos objetos da aplicação e dos meta-objetos que implementam as ações de tolerância a falhas, uma vez que podem ser independentemente desenvolvidos, testados e verificados;

c) Biblioteca de classes de objetos que implementam diferentes estratégias de tolerância a falhas, que podem ser diretamente instanciados como meta-objetos associados a objetos da aplicação, ou podem ser redefinidos para melhor adequação aos objetivos da aplicação.

Além da sua aplicação na construção de software tolerante a falhas, outros usos também são sugeridos para objetos tolerantes a falhas: testes de componentes e transição de versões.

Como técnica tardia de depuração, objetos tolerantes a falhas podem ser usados com diversos objetivos: como técnica suplementar de testes [ABB90] , como técnica para viabilizar testes de sistemas distribuídos que se caracterizam por uma grande diversidade de estados da computação resultantes de combinações de componentes concorrentes [KEL91], ou ainda, para selecionar componentes para uma aplicação, buscados em um repositório de objetos reusáveis. Entre os componentes testados, pode-se selecionar o mais adequado à aplicação.

Capítulo 5

MODELAGEM DE META-OBJETOS

Detalha a arquitetura reflexiva descrita no Capítulo 4, descrevendo as classes de objetos do *framework* MOTF no método HOOD.

5.1 O enfoque da modelagem

O objetivo deste capítulo é a descrição dos componentes dos Meta-Objetos para Tolerância a Falhas, permitindo que a sua leitura forneça elementos suficientes para que estes componentes sejam alvo de projeto mais detalhado para que possam ser implementados em uma linguagem de programação orientada a objetos. A modelagem descreve as classes e suas associações, bem como sua utilização na arquitetura reflexiva.

A linguagem C++ foi utilizada para a implementação de alguns protótipos e exemplos mostrados no capítulo 4 e no presente capítulo, juntamente com o protocolo de meta-objetos oferecidos pela extensão OpenC++. Os protótipos e exemplos foram desenvolvidos para ilustrar idéias e testar teorias; são de concepção simples e de implementação descompromissada com eficiência e generalidade. A seção 5.3.3 mostra um exemplo de expansão de código com essas características.

Outrossim, o âmbito deste trabalho ultrapassa as particularidades de uma linguagem de programação e as limitações e peculiaridades de uma implementação específica. Pretende, sobretudo, demonstrar que o modelo de programação orientada a objetos possui características próprias que simplificam a tarefa de desenvolvimento de aplicações tolerantes a falhas e que a técnica de reflexão computacional contribui para essa simplificação, ao oferecer abstração dos mecanismos e estender a semântica da linguagem de programação.

Ao procurar delimitar a abrangência deste capítulo, a seguinte definição de Booch [Boo94] serviu de diretriz:

> Os limites entre análise e projeto são confusos, embora o foco de cada um deles seja deveras distinto. Na análise, procuramos modelar o mundo, *descobrindo* as classes e modelos que formam o vocabulário do domínio do problema, e, no projeto, nós *inventamos* as abstrações e mecanismos que fornecem o comportamento que este modelo requer.

A prioridade deste capítulo é o processo de análise. Embora prioritário, este enfoque certamente não é totalmente isento de modelagens e soluções apoiadas em aspectos semânticos da linguagem de prototipação. Na descrição das classes através de ODS - Object Description Skeleton[1] desce

---

[1] Notação HOOD.

a nível de projeto de programas, sugerindo construções apoiadas na semântica da linguagem C++ e soluções de reflexão computacional dadas por OpenC++.

5.2 Desenvolvimento de aplicações tolerantes a falhas

Os serviços oferecidos pelo MOTF podem ser classificados em duas categorias, que dividem em duas fases o processo de desenvolvimento de aplicações tolerantes a falhas:

I - Serviços de configuração da aplicação tolerante a falhas; e,

II - Serviços de tolerância a falhas implementados na aplicação.

Os primeiros interagem com o usuário (programador da aplicação) e tem os seguintes objetivos:

a) Identificar os objetos tolerantes a falhas da aplicação;

b) Selecionar e particularizar as técnicas de tolerância a falhas usadas pela aplicação;

c) Relacionar os objetos tolerantes a falhas com as respectivas técnicas;

d) Configurar a aplicação, incluindo as classes de meta-objetos correspondentes às técnicas utilizadas.

A Fase II prepara a aplicação tolerante a falhas, reunindo os objetos da aplicação com os meta-objetos associados, estruturados a partir de diversas (meta) classes descendentes da classe MOTFs-Serviços. O *framework* MOTF, portanto, abrange as classes de configuração da aplicação e as classes dos meta-objetos que fornecem os diversos serviços durante a execução da aplicação. As primeiras correspondem aos protocolos de interface com o usuário, não participando do código da aplicação e as demais referem-se aos protocolos de interface com os objetos tolerantes a falhas e os serviços de tolerância a falhas.

O *framework* MOTF depende de informações da aplicação; a aplicação é considerada um *usuário* dos serviços e os objetos da aplicação são considerados como *objetos participantes* de alguma técnica de tolerância a falhas. A figura 5.1 mostra a classe que representa o *framework* MOTF, com suas interfaces genéricas **Usuário** e **ObjetosParticipantes**, seus componentes **Motf_Interface**, **Motf_Configurador** e **Motfs_Serviços**, bem como seu

relacionamento com a classe abstrata **Ambiente**. A interface genérica Usuário representa todas as diretivas e componentes fornecidos pelo programador para a configuração da aplicação e a interface genérica ObjetosParticipantes identifica os objetos do domínio da aplicação que devem ser tolerantes a falhas, incluindo os componentes redundantes.

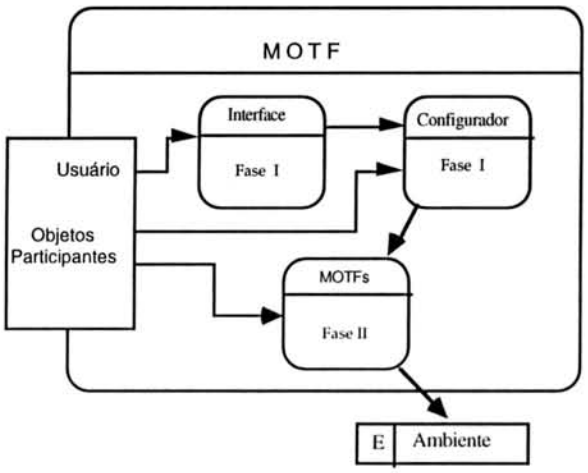


Figura 5.1 - Arquitetura básica do MOTF

A classe abstrata identificada por E ('Environment') representa, no método HOOD, a interface de outro objeto a ser usado pelo sistema, não sendo parte integrante do sistema para fins de detalhamento de projeto. Na arquitetura MOTF, corresponde à classe Ambiente, a qual encapsula e representa todos os componentes que propiciam a interface com o ambiente de suporte, para serviços tais como concorrência e distribuição de objetos.

5.3 A configuração de aplicações

Os serviços de configuração procuram auxiliar o programador da aplicação, ao gerenciar a inserção na aplicação de componentes responsáveis pelas características de tolerância a falhas.

Porém, a existência de uma biblioteca de classes possibilita ao programador mais experiente utilizar diretamente as classes de serviços, adequando-as às necessidades específicas da aplicação. As classes a serem inseridas na aplicação também admitem especializar propriedades, de forma a obter variantes das técnicas originais.

5.3.1 Classes Motf- Fase I

Estes serviços são executados sob a responsabilidade de duas classes:

(a) **Classe Motf_Interface:** representa a interface genérica do *framework* MOTF.

Esta classe idealmente deve propiciar uma interface gráfica com o usuário para a obtenção de diretivas, visto que a sua missão é oferecer serviços de tolerância a falhas e buscar informações para a configuração. Estas informações incluem os objetos que devem ser tornados tolerantes a falhas, os demais componentes redundantes e a estratégia de tolerância a falhas a ser adotada. No contexto deste trabalho, as diretivas são fornecidas sob forma de comentários no texto fonte da aplicação. Servem como base para a expansão de código através de um pré- processador.

(b) **Classe Motf_Configurador:** utiliza as diretivas de configuração da aplicação tolerante a falhas para selecionar e particularizar os serviços de tolerância a falhas a serem incorporados à aplicação.

Esta classe implementa os serviços de tolerância a falhas na aplicação utilizando bibliotecas de classes e pré-processadores[2]. As bibliotecas de classes compreendem todas as classes do *framework* MOTF e as classes de meta-objetos. As primeiras implementam os serviços específicos de tolerância a falhas e as demais implementam os mecanismos que suportam a reflexão computacional.

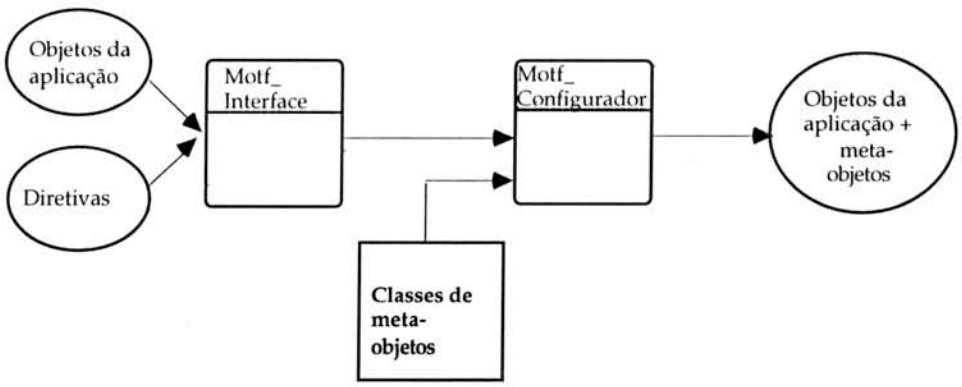


Figura 5.2 - Configuração da aplicação

---

[2] Dois pré-processadores: de diretivas de Motf e diretivas de OpenC++.

No processo de configuração são selecionadas as metaclasses correspondentes ao serviço desejado e associadas aos objetos tolerantes a falhas. A figura 5.2 esquematiza a execução da Fase I - Serviços de configuração da aplicação tolerante a falhas. Os métodos e técnicas de tolerância a falhas especificados pelo usuário são transformados, pelo Configurador, em diretivas para os MOTFs-Serviços. As classes dos meta-objetos, hierarquicamente organizadas, permitem a particularização de meta-objetos (nível $\mathcal{D}_1$) de acordo com as diretivas recebidas e o conjunto de objetos da aplicação (nível $\mathcal{D}_0$)[3].

5.3.2 Diretivas

As diretivas Motf são inseridas no texto fonte da aplicação, segundo o formato básico: //<classe Motf> <args>, onde <classe Motf> corresponde ao metaobjeto associado ao objeto tolerante a falhas.

A associação a um metaobjeto é feita de acordo com as diretivas, abrangendo diversos casos, conforme esquematizado na Tabela 5.1.

Os argumentos <args> são codificados de acordo com a convenção a seguir:

OC: Classe do objeto tolerante a falhas
MC: Nome do método crítico
N: Número de objetos participantes
PC: Classes dos objetos participantes do grupo

Tabela 5.1 - Diretivas MOTF

Diretiva MOTFs	Técnica	Objetos participantes
//Motf_RB<OC,MC,N,PC,TA>	Blocos de recuperação	Objeto interface, alternativas e teste de aceitação
//Motf_NVP<C, OC,MC,N,PC>	N-Versões Sem distribuição	Objeto interface Classes diversitárias
//Motf_NVP<D, OC,MC,N,PC>	N-Versões Com distribuição	Objeto interface Classes diversitárias
//Motf_RA<OC,N>	Replicação ativa	Classe do objeto Número de réplicas
//Motf_RP<OC,N>	Replicação passiva	Classe do objeto Número de réplicas

[3] Ver Cap. 4, Seção 4.5.

5.3.3 Exemplo de expansão de código

O exemplo abaixo ilustra uma aplicação de programação em n-versões onde o objeto tolerante a falhas é definido pela classe Interpola, e o método crítico é denominado de InterpolaValor. O usuário fornece as classes diversitárias denominadas Newton, Linear e Lagrange, todas descendentes da classe Interpola e a diretiva:

//Motf _NVP(C,Interpola,InterpolaValor,3,Newton,Lagrange, Linear)

O configurador passa a executar as seguintes ações:

1) Incluir os arquivos correspondentes às classes:

```
#include "metaobj.h"          /* OpenC++ */
#include "divmetaobj.h"       /* MOTF- classes diversitárias */
#include "grupostf.h"         /* MOTF- grupos */
```

2) Incluir a diretiva de método reflexivo na classe Interpola

```
Class Interpola { /* classe do objeto tolerante a falhas */
public:
//MOP reflect
        double InterpolaValor(....)
};
```

3) Incluir a diretiva de associação da classe Interpola ao meta-objeto da classe Motf_NVP

```
//MOP reflect class Interpola:Motf_NVP;
```

4) Incluir a estrutura de grupo: um grupo de objetos é instanciado, com o nome de GupoInterpola, composto por três classes de objetos. A criação de um grupo é expandida para incluir a instanciação de seus membros:

```
Motf_Grupo GrupoInterpola(3,Newton,Lagrange,Linear);
internewton = new Newton;
GrupoInterpola.insere(Newton,internewton);
interlagrange = new Lagrange;
GrupoInterpola.insere(Lagrange,interlagrange );
interlinear = new Linear;
GrupoInterpola.insere(Linear,interlinear);
```

A figura 5.3 mostra o código simplificado do programa, após a expansão. O texto em negrito refere-se ao código adicional que implementa a programação em n-versões e a reflexão computacional, usando a semântica de execução para melhor legibilidade do texto fonte.

```
Class Interpola { /* classe do objeto tolerante a falhas */
public:
//MOP reflect
        double InterpolaValor(....)
};
/* classes dos objetos diversitários */
Class Newton {
.....
public:
        double InterpolaValor(....)
};
Class Linear {
.....
public:
        double InterpolaValor(....)
};
Class Lagrange {. .....
public:
        double InterpolaValor(....)
.....};
/* classe do metaobjeto */
Class Motf_NVP: public Metaobject
{
public:
        void Meta_methodCall(....) {
/* intercepta/executa chamada ao método reflexivo*/
Meta_MethodCall(mid,args,rep);
/* executa objetos diversitários */
Y_newton=internewton.InterpolaValor(args,rep);
Y_lagrange=interlagrange.Interpolavalor(args,rep);
Y_linear=interlinear.InterpolaValor(args,rep);
/* vota e devolve resultado */
Y= Votador(Y_newton,Y-lagrange,Y_linear);
return Y;
};
/* associa meta-objeto da classe motf_NVP à classe Interpola */
//MOP reflect class Interpola:Motf_NVP;
/* principal*/
void main( )
{.  .....
refl_Interpola inter;          /* instancia objeto reflexivo */
Y= inter.InterpolaValor(...)   /* chama o método crítico */
.....}
```

Figura 5.3 - Implementação de programação em n-versões

5.3.4 Classes Motf - Fase II

Essas classes são representadas no nível mais alto da hierarquia pela classe Motfs, usada para definir os comportamentos básicos de objetos tolerantes a falhas. As demais classes básicas de serviços são:

Grupo de objetos:	Classe Motf_GrupoTF
Objetos reflexivos:	Classe ObjRefl
Programação diversitária:	Classe Motf_PD
Programação em n-versões:	Classe Motf_NVP
Blocos de recuperação:	Classe Motf_RB
Replicação de objetos:	Classe Motf_REP
Replicação ativa:	Classe Motf_RA
Replicação passiva:	Classe Motf_RP

A estrutura abaixo (figura 5.4.) mostra a hierarquia de herança das principais classes de serviços.

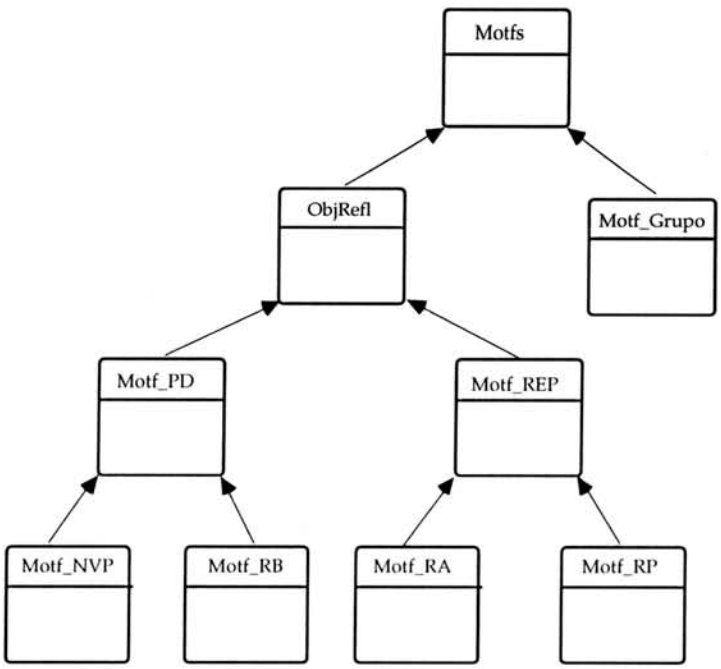


Figura 5.4 - Hierarquia de herança

Esta hierarquia reflete a unificação das diferentes estratégias de tolerância a falhas, centralizando o foco de atenção de todas as técnicas na classe representante de um objeto reflexivo tolerante a falhas (ObjRefl). Desta classe descendem todas as classes que implementam as técnicas de

redundância. O outro conceito central, o de grupo de objetos é destacado como um dos principais componentes; porém, sua associação com as demais classes não é estabelecida por herança e sim *por uso*. Sua importância será realçada nas seções a seguir, onde são definidas as responsabilidades das classes básicas, identificados seus componentes e suas associações.

5.4 Classe MOTFs

5.4.1 Responsabilidades

É uma classe abstrata que define os elementos constantes do ambiente, como os estados da computação (ex. normal, exceção, falha) e as interfaces aos serviços de tratamento de exceções, salvamento de estados e recuperação de objetos. As exceções aqui mencionadas são as propagadas ao cliente do objeto tolerante a falhas, na impossibilidade de fornecimento do serviço solicitado por nenhum dos objetos redundantes.

Os tipos básicos mantém a uniformidade das informações das classes descendentes e as interfaces, definidas como virtuais, devem ser posteriormente implementadas nas suas classes descendentes.

5.4.2 Descrição da classe

CLASS **MOTFs**
DESCRIPTION Esta classe, que corresponde ao nível mais alto da hierarquia, é usada para definir os comportamentos básicos de objetos tolerantes a falhas.
PROVIDED_INTERFACE
TYPES [-- Tipos públicos definidos pela classe.]
EstadoExec {normal, executando, morreu, abortado...}
Condição {ativo, inativo}
Resultado (maioria, maioria_absoluta, sem_resposta...}
OPERATIONS [-- Métodos públicos definidos pela classe]
virtual SalvaEstadoLocal(objeto)
DESCRIPTION Salva o estado de um objeto local
virtual SalvaEstadoRemoto(objeto)
DESCRIPTION Salva o estado de um objeto remoto (processo)
virtual RecuperaMembroLocal(objeto)
DESCRIPTION Cria nova instância de um objeto local, restaurando estado.
virtual RecuperaMembroRemoto(objeto)
DESCRIPTION Cria nova instância de um objeto remoto, restaurando estado.
Clone(objeto1, objeto2)
DESCRIPTION cria um novo objeto em objeto2 e faz uma cópia do objeto1 no objeto2, por atribuição
EXCEPTIONS [-- Exceções que podem ser transmitidas a outros objetos.]
virtual Erro_Interface
DESCRIPTION O objeto/método não reconhece a mensagem.

virtual Erro_Local
DESCRIPTION Os objeto não podem ser executados localmente.
virtual Erro_Remoto
DESCRIPTION Falha na distribuição de objetos.
virtual Outro_Erro
DESCRIPTION Todos os demais erros de execução.
END_CLASS **MOTFs**

5.5 Classe MOTF_GrupoTF

Os objetos são agrupados na classe MOTF_GrupoTF. Esta classe representa o nível mais alto da hierarquia[4] que organiza uma interface comum a um grupo de objetos. Todos os membros do grupo obedecem à mesma estratégia de tolerância a falhas, possuem o mesmo tipo e podem ser réplicas de um mesmo objeto ou objetos diversos de igual protocolo. Um grupo de objetos tolerantes a falhas oferece os seguintes serviços básicos:

a) Organizar os membros do grupo: criar um grupo de objetos; desativar um grupo de objetos; inserir novos membros no grupo de objetos; retirar objetos do grupo.

b) Dar informações sobre os membros do grupo: número de membros em atividade, nomes dos membros, resultados da monitoração.

c) Disseminar mensagens a todos os membros do grupo: estes comportamentos são obtidos através da classe Mensageiro.

d) Coletar resultados da execução dos membros do grupo.

f) Exibir/gravar os resultados da execução dos membros do grupo.

As responsabilidades são distribuídas em diversas classes associadas que encapsulam os serviços (figura 5.5).

[4]Em C++, corresponde a um conjunto de classes reunidas em um arquivo 'header'.

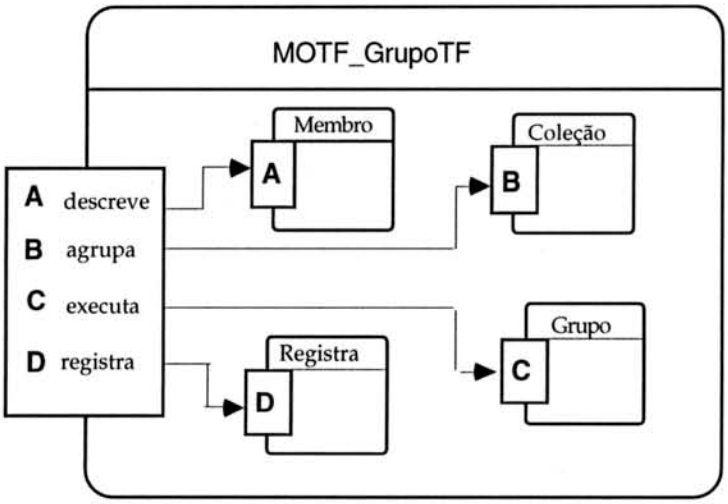


Figura 5.5 - Diagrama da classe MOTF_GrupoTF

5.5.1 Definição de responsabilidades

Membro: Um componente atômico de um grupo é um *membro*. Cada membro é um objeto participante (alternativa, versão ou réplica) de uma técnica de tolerância a falhas.

Um membro é definido por uma estrutura de dados contendo as seguintes informações: nome da classe, nome do objeto, prioridade e condição do membro (ativo, inativo). O nome da classe é obtido através da diretiva de configuração (Ver tabela 5.1), o nome da instância é criado localmente e permanece invariante, a prioridade estabelece a ordem de execução e a condição indica se o objeto deve participar da próxima execução. Um membro é registrado como ativo quando ele é instanciado como participante de uma técnica de TF; torna-se inativo ao abandonar, temporaria ou definitivamente, o grupo de objetos.

Sobre um membro podem ser realizadas as seguintes operações: instanciação de um membro e alteração de sua prioridade e condição.

Coleção: Os membros são agrupados em *coleções*, sendo que uma coleção compreende todos os membros participantes de uma mesma técnica de tolerância a falhas.

Uma coleção é definida por uma estrutura de dados contendo as informações necessárias à execução de cada membro[5]. Uma coleção pode ser

[5]Em C++, corresponde a um apontador para a instância.

composta por objetos da aplicação ou por meta-objetos associados a objetos da aplicação.

Várias operações podem ser realizadas sobre uma coleção de membros:

a) uma coleção é dinâmica, visto que membros podem ser inseridos, retirados, ativados e desativados;

b) uma coleção pode receber solicitações de informações sobre seus membros, a saber: lista de todos os membros e informações individuais sobre membros.

Grupo: Uma coleção de objetos pode ser executada em diversos modos: sob demanda, em seqüência, em paralelo ou de modo distribuído. O modo de execução é ditado pela técnica de tolerância a falhas.

- Execução sob demanda: um determinado membro do grupo é executado.
- Modo seqüêncial: todos os membros ativos são executados seqüencialmente, em ambiente centralizado, na ordem em que se encontram registrados na tabela de membros.
- Modo paralelo: todos os membros ativos são executados concorrentemente, em ambiente centralizado.
- Modo distribuído: todos os membros ativos são executados concorrentemente, em ambiente distribuído.

A classe Motf_Grupo, esquematizada na figura 5.6, fornece os diversos protocolos de execução de coleções de objetos e sua associação com os objetos da aplicação.

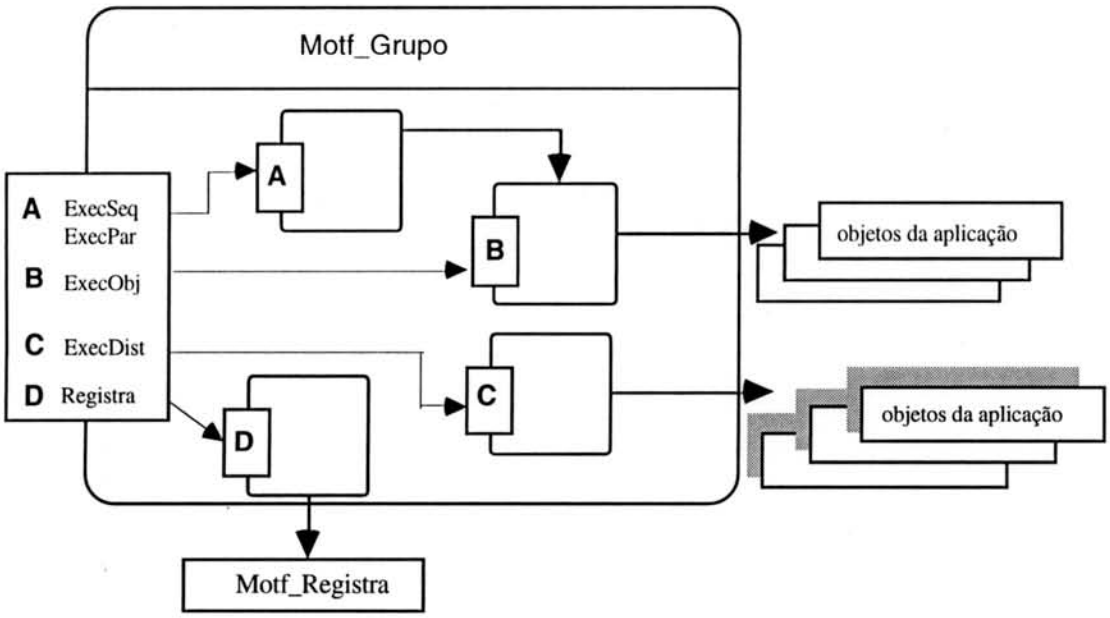


Figura 5.6 - Diagrama da classe Motf_Grupo

Em ambientes centralizados, os objetos da aplicação não necessitam de controle individual; em ambientes distribuídos, cada um dos objetos é controlado por um meta-objeto individual, representado por uma sombra.

Ainda em ambientes distribuídos, os protocolos de distribuição são interpretados por um Mensageiro, como a seguir descrito.

Mensageiro: O objeto desta classe encarrega-se de disseminar as mensagens a todos os membros do grupo e corresponde ao meta-nível D_2 da arquitetura reflexiva. Utiliza as primitivas do sistema operacional, especializando assim os serviços de distribuição. Possui as seguintes responsabilidades:

a) Instalação/execução de um membro em um processador remoto.

b) Localização: mantém informações sobre a localização de cada membro do grupo;

c) Estado: mantém informações sobre o estado de execução de cada membro do grupo.

d) Replicação distribuída de objetos.

O mensageiro não possui qualquer ação de tolerância a falhas; apenas encarrega-se da execução dos componentes do grupo e fornece informações sobre a execução.

Sendo a classe que faz a interface com as primitivas do sistema operacional ou sistema de distribuição, esta classe é definida como classe abstrata, fornecendo acesso a diferentes protocolos de execução distribuída. Em especial, a ação (d) depende das facilidades de replicação de grupos de objetos oferecidas pelo sistema.

5.5.2 Descrição da classe Membro

Class **Motf_Membro**
DESCRIPTION Esta classe define a estrutura de dados usada para descrever as informações estáticas sobre cada objeto.
PROVIDED_INTERFACE
TYPES [-- A definicao de tipos públicos é apenas sugerida, sem detalhes específicos de implementação.]
ClasseObj
DESCRIPTION Nome da classe do objeto. Pode ser um apontador constante.
NomeObj
DESCRIPTION Nome local do objeto. Pode ser um string; ver observação a seguir.
Prioridade
DESCRIPTION Indicador de prioridade de execução.
Membro
DESCRIPTION Estrutura de dados contendo: ClasseObj, NomeObj, Prioridade e Condição (ativo,inativo)
OPERATIONS [-- Métodos públicos definidos pela classe]
Insere_Classe
DESCRIPTION Coloca na tabela a classe do objeto.
Insere_Nome
DESCRIPTION Coloca na tabela o nome do objeto.
Nome_Classe
DESCRIPTION Fornece o nome da classe.
Nome_Objeto
DESCRIPTION Fornece o nome do membro (instância)
Ativo
DESCRIPTION Informa se o membro está ativo
AlteraCond
DESCRIPTION Altera a condição de ativo/passivo ou vice-versa
END_CLASS **Motf_Membro**

Observação: o nome do membro é utilizado para documentação, podendo ser utilizado para informações sobre a execução dos objetos.

5.5.3 Descrição da Classe Motf_Coleção

Class **Motf_Coleção is Motf_Membro**
DESCRIPTION Esta classe define a estrutura de dados que organiza todos os membros participantes de uma mesma técnica de tolerância a falhas e define uma interface comum a um grupo de objetos.
PROVIDED_INTERFACE
TYPES [-- Tipos públicos definidos pela classe.]
Coleção

DESCRIPTION Estruturas de dados contendo informações sobre a estrutura dos membros. Vetor do tipo Membro.

MembroObj

DESCRIPTION Estrutura de dados contendo os endereços dos membros do grupo (instâncias) em ambientes centralizados. Vetor de apontadores de objetos.

MembroProc

DESCRIPTION Estrutura de dados contendo os endereços dos membros do grupo (instâncias) em ambientes concorrentes. Cada membro é considerado um processo.

OPERATIONS [-- Métodos públicos definidos pela classe]

Motf _Agrupa(quant)

DESCRIPTION Construtor que recebe a quantidade de objetos do grupo e inicializa a coleção e o vetor de instâncias.

virtual Motf_Insere(ClassObj, NomeObj, Instância)

DESCRIPTION Recebe o nome da classe do objeto e o nome do objeto e endereço de sua instância e o insere na tabela de membros. Confere a informação de classe e de nome.

Grupo_Retira(indice)

DESCRIPTION Recebe o índice correspondente à posição do objeto na tabela de membros e torna inativo o objeto.

Grupo_Tam

DESCRIPTION Informa o número de membros ativos do grupo.

Grupo_Lista

DESCRIPTION Fornece uma lista de todos os membros do grupo.

GrupoInfo(NomeObj)

DESCRIPTION Informa a situação de um membro do grupo.

Not_Grupo

DESCRIPTION Destrutor que elimina um grupo.

END_CLASS **Motf_Coleção**

Observações: Pode ser definido apenas um tipo de dados para conter as informações necessárias à execução dos membros. O tipo de dados é idealizado como uma abstração dos endereços de execução. Em ambientes centralizados, pode ser definido com apontadores para as instâncias. Em ambientes distribuídos, outras informações se tornam necessárias.

5.5.4 Descrição da classe Motf_Grupo

CLASS **Motf_Grupo** IS Motf_Coleção

DESCRIPTION Esta classe fornece as informações dinâmicas e os protocolos de execução dos membros de um grupo.

PROVIDED_INTERFACE

TYPES [-- Tipos públicos definidos pela classe.]

Resultados

DESCRIPTION Estrutura de registro que contém os resultados da execução de cada membro do grupo. É vinculada à estrutura de grupos, através do apontador. Contém as seguintes informações: localização (local, remota), situação (normal, morreu, abortado, outro), hora (inicial e final) e apontador para o resultado da última execução.

OPERATIONS [-- Métodos públicos definidos pela classe]

ExecObj (args,result)

DESCRIPTION Recebe os argumentos da chamada e executa o proximo membro do grupo por ordem de prioridade. Controla a execução, coleta os resultados e os coloca na tabela de resultados.

GrupoExecSeq (args,result)

DESCRIPTION Recebe os argumentos da chamada e executa os membros do grupo em seqüência. Controla a execução, coleta os resultados e os coloca na tabela de resultados.

GrupoExecPar (args,result)

DESCRIPTION Recebe os argumentos da chamada e executa os membros do grupo em paralelo. Controla a execução, coleta os resultados e os coloca na tabela de resultados.

GrupoExecDist(nome_do_grupo)

DESCRIPTION Recebe os argumentos da chamada e executa todos os membros do grupo de forma distribuida. Controla a execução, coleta os resultados e os coloca na tabela de resultados.

EXCEPTIONS [-- Exceções que podem ser transmitidas a outros objetos.]

FalhaMembro

DESCRIPTION Todos os tipos de exceção sinalizadas pelo membro em execução. Ver Observação abaixo

REQUIRED_INTERFACE

CLASS [-- Lista de outras classes Motf usadas]

Mensageiro

DESCRIPTION Classe usada para objetos distribuídos.

END_CLASS **Motf _Grupo**

Observação: As exceções aqui referidas são aquelas propagadas de forma dinâmica: uma exceção capturada em um objeto durante a execução de um serviço é propagada ao objeto solicitante do serviço. Neste caso, o solicitante do serviço é o metaobjeto associado ao servidor, uma vez que as requisições são desviadas para o meta-nível.

Os mecanismos de tratamento de exceções possuem diferentes interpretações e implementações nas linguagens de programação [LIS94]. Uma interpretação bastante adequada às necessidades do *framework* MOTF é a nível de comando, conforme sugerida por Carvalho [CAR92]:

```
object-name <- operation_name [(argument..)]
exception
    when exception-name [(parameter...)]
        then statement,....
end exception;
```

Em C++ a construção acima é implementada usando do seguinte esquema [KOE90]:

```
try
    {
      ExecObj(i)...
    }
catch(...)
{
     throw exceção....
}
```

Este tratamento a nível de comando permite preservar o ambiente no qual a solicitação de operação foi feita; se a operação sinalizar uma exceção ela é localmente tratada. No domínio de tolerância a falhas, a sinalização de uma exceção é uma indicação de comportamento anormal e que providências devem ser tomadas para a continuidade do serviço, como por exemplo, executar uma outra versão da operação falha.

É importante notar que estas considerações referem-se a exceções propagadas ao meta-nível. No nível base, os objetos da aplicação podem (e devem) ser tornados mais *robustos* pela inclusão de tratadores locais.

5.5.5 Descrição da classe Motf_Registra

CLASS **Motf_Registra** is **Motf_Coleção**
DESCRIPTION Esta classe fornece os serviços de registro dos dados coletados durante o processo de execução de grupos de objetos.
PROVIDED_INTERFACE [-- Descrição dos recursos fornecidos a outros objetos.]
TYPES [-- Tipos públicos definidos pela classe.]
Registra_Grupo
DESCRIPTION Estrutura que organiza as informações estáticas sobre o grupo de objetos, para fins de exibição e gravação de resultados da execução.
Registra_Versoes
DESCRIPTION estrutura de registro contendo informações sobre o resultado da execução: id_execução, quantidade de versões, argumentos de chamada, resultado final, e, para cada versão: resultado, participação da versão no resultado final (sim,não) e indicação de ocorrência de execução (normal, erro, tipo de erro).
OPERATIONS [-- Métodos públicos definidos pela classe]
Motf_Registro(grupo)
DESCRIPTION Método construtor que inicializa a tabela Registra_Grupo, com os dados estáticos.
GrupoInfo
DESCRIPTION Fornece informações sobre a última execução de todos os membros do grupo (nome, resultado da execução).
ObjetoInfo (nome_do_objeto)
DESCRIPTION Fornece informações sobre o objeto (nome, resultado da execução).
Grupo_grava
DESCRIPTION Grava informações sobre a última execução de todos os membros do grupo.

EXCEPTIONS [-- Exceções que podem ser transmitidas a outros objetos.]
ENVIRONMENT_CLASS [-- Lista de outras classes usadas]
DESCRITION Utiliza bibliotecas do sistema para gravação de arquivos e exibição de resultados.
EXCEPTIONS [-- Lista de exceções capturadas]
DESCRIPTION Todas as exceções referentes a entrada/saida.
END_CLASS **Motf _Registra**

5.5.6 Descrição da classe Motf_Mensageiro

CLASS **Motf_Mensageiro**
DESCRIPTION O mensageiro transforma chamadas locais em chamadas remotas de objetos. Encarrega-se de disseminar as mensagens a todos os membros do grupo, por requisição do objeto Motf_Grupo. A cada objeto do grupo corresponde um mensageiro. Implementa distintos mecanismos de comunicação.
PROVIDED_INTERFACE [-- Descrição dos recursos fornecidos a outros objetos.]
TYPES [-- Tipos públicos definidos pela classe.]

CONSTANTS [-- Constantes públicas definidas pela classe.]
OPERATIONS [-- Métodos públicos definidos pela classe]
DESCRIPTION Construtor que inicializa a tabela de distribuição de objetos
ExecRPC(nome_do_objeto, args, resp, status)
DESCRIPTION Recebe os argumentos da chamada e executa o objeto via Remote Procedure Call. Controla a execução, coleta os resultados e os devolve.
ExecBSD(nome_do_objeto, args, resp, status)
DESCRIPTION Recebe os argumentos da chamada e executa o objeto via BSD sockets. Controla a execução, coleta os resultados e os devolve.
Timer(tempo_max)
DESCRIPTION Controla o tempo de execução dos objetos
EXCEPTIONS [-- Exceções que podem ser transmitidas a outros objetos.]
DESCRIPTION Todas as exceções referentes a execução distribuída disponíveis no sistema.
REQUIRED_INTERFACE [-- Descrição dos recursos de outros objetos.]
DESCRIPTION Requer acesso a primitivas de distribuição do sistema operacional, para execução concorrente/distribuída de objetos.
CLASS [-- Lista de outras classes Motf usadas]
ENVIRONMENT_CLASS [-- Lista de outras classes usadas]
EXCEPTIONS [-- Lista de exceções capturadas]
DESCRIPTION Utiliza as exceções sinalizadas pelo sistema de suporte

INTERNALS [-- Descreve os recursos protegidos e privativos da classe]
TYPES [-- Tipos definidos pela classe.]
Objeto_endereco
DESCRIPTION Mantém os endereços dos nodos de execução de cada membro do grupo, para execução concorrente
Objeto_estado
DESCRIPTION Mantém informações sobre o estado de execução dos processos
CONSTANTS [-- Constantes definidas pela classe.]
OPERATIONS [-- Métodos privativos definidos pela classe]
ObjectCall (nome_do_objeto)
DESCRIPTION Inicia a execução de um objeto
ObjectSuspend(nome_do_objeto)
DESCRIPTION Suspende a execução de um objeto

ObjectCheck(nome_do_objeto)
DESCRIPTION Verifica o estado de um método.
END_CLASS **Motf_Mensageiro**

5.6 Objetos Reflexivos Tolerantes a Falhas

Na arquitetura reflexiva de MOTF, a inclusão de redundância na aplicação se baseia no conceito de *objeto reflexivo tolerante a falhas*. Este é um objeto da aplicação que recebe a solicitação de execução de um serviço crítico que implica em redundância.

5.6.1 Responsabilidades

Um objeto reflexivo tolerante a falhas é associado a um meta-objeto, com capacidade de:

- interceptar chamadas a métodos reflexivos do objeto referente;

- ter acesso aos argumentos de chamada;

- ter acesso às variáveis reflexivas de seu referente;

- ter acesso a informações estruturais a respeito de seu referente: sua classe, o nome do método chamado e nome de suas varáveis reflexivas.

- executar, em lugar do cliente original, métodos reflexivos de seu referente.

Essas capacidades são herdadas da classe MetaObj (OpenC++) e particularizadas para a implementação de serviços de tolerância a falhas através dos seguintes métodos:

Entrada: fornece os argumentos de chamada;

Saida: fornece a resposta ao cliente;

InfObj: exibe informações sobre o objeto reflexivo;

Esses métodos são públicos e virtuais, permitindo a sua redefinição para sua adequação aos objetos da aplicação.

5.6.2 Descrição da classe ObjRefl

CLASS **ObjRefl** IS MetaObj
DESCRIPTION Herda da classe MetaObj os protocolos de reflexão computacional e os particulariza para os objetos da aplicação.
PROVIDED_INTERFACE [-- Descrição dos recursos fornecidos a outros objetos.]
OPERATIONS [-- Métodos públicos definidos pela classe]

virtual Entrada (ArgPac)
DESCRIPTION Fornece os argumentos de chamada do método reflexivo, em um objeto da classe ArgPac.
virtual Saida (ArgPac)
DESCRIPTION Fornece o resultado da execução do método reflexivo, em um objeto da classe ArgPac.
virtual InfObj (ArgPac)
DESCRIPTION Exibe informações sobre o objeto reflexivo: sua classe e seu nome.
END_CLASS **ObjRefl**

Observações: Um objeto reflexivo tolerante a falhas é sempre um objeto *local*, mesmo em caso de execução distribuída. Na aplicação, um objeto tolerante a falhas é associado a uma das seguintes classes de serviços: Motf_NVP (programação em n-Versões), Motf_RB (blocos de recuperação), Motf_RA (replicação ativa) ou Motf_RP (replicação passiva).

5.7 Classe Motf_PD

Esta (meta) classe de técnicas de programação diversitária abrange os serviços para a implementação de dois métodos clássicos de tolerância a falhas: a programação em n-versões e blocos de recuperação. Embora distintas na sua concepção, estas duas técnicas observam alguns comportamentos similares, visto que ambas utilizam múltiplas versões, requerem um único resultado final, exigem monitoração da execução das versões e podem coletar dados estatísticos durante a execução.

5.7.1 Responsabilidades

Estes comportamentos similares são descritos em uma classe ancestral, comum a ambas as técnicas. As estruturas de dados necessárias ao suporte das atividades comuns e os métodos genéricos são transmitidos estáticamente por herança, sendo particularizados somente os comportamentos distintos, como a execução seqüencial ou paralela, os algoritmos de votação ou aceitação e o estabelecimento de pontos de recuperação.

Os serviços básicos oferecidos por esta classe são:

a) reflexão computacional;

b) preparação de um grupo de objetos diversificados;

c) preparação dos dados para a execução das versões;

d) tratamento das versões faltosas.

As responsabilidades são distribuídas em diversas classes associadas que encapsulam os serviços. Um exemplo, neste caso, é a definição de um objeto reflexivo que herda o protocolo de meta-objetos. A figura 5.7 mostra o diagrama dos serviços básicos oferecidos pela classe Motf_PD, os métodos que organizam esses serviços e o fluxo de dados, representados pelas setas (O-->).

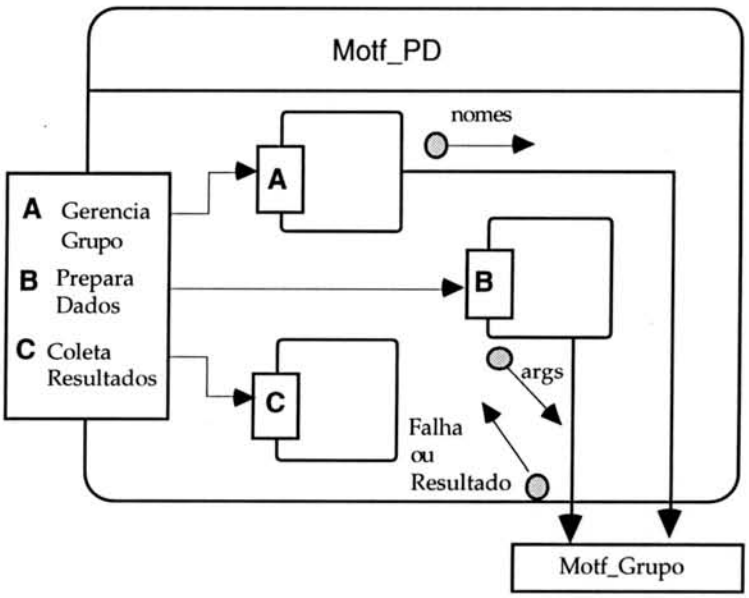


Figura 5.7 - Objetos de Programação Diversitária

5.7.2 Descrição da classe Motf_PD

CLASS **Motf _PD** is ObjRefl
DESCRIPTION Classe abstrata que implementa as estruturas de dados e métodos comuns às técnicas de programação em n-versões e blocos de recuperação
PROVIDED_INTERFACE [-- Descrição dos recursos fornecidos a outros objetos.]
OPERATIONS [-- Métodos públicos definidos pela classe]
Gerencia_Grupo
DESCRIPTION É decomposta em um conjunto de outras operações internas. Permite a criação, reconfiguração e destruição de grupos de objetos diversificados.
Prepara_Dados
DESCRIPTION Normaliza os argumentos de execução dos objetos do grupo.
Coleta_Resultados
DESCRIPTION Recebe os resultados da execução dos membros do grupo.
EXCEPTIONS [-- Exceções que podem ser transmitidas a outros objetos.]
Falha_PD
DESCRIPTION Indica que não foi possível fornecer uma resposta ao cliente do objeto tolerante a falhas. Pode ser decomposta para explicitar o tipo de

```
                falha: sem aceitação de resultados, falhas na restauração de estado,
                etc.
REQUIRED_INTERFACE [-- Descrição dos recursos de outros objetos.]
        CLASS [-- Lista de outras classes Motf  usadas]
        Motf_Grupo
        ENVIRONMENT_CLASS [-- Lista de outras classes usadas]
        Mensageiro
END_CLASS Motf_PD
```

5.8 Classe Motf_NVP

O Motf_NVP deve prover um ambiente de execução para apoiar computações múltiplas. As computações múltiplas são implementadas por $n \geq 3$ objetos de igual funcionalidade, que possuem um objetivo comum: produzir um conjunto de n resultados, derivados de um mesmo conjunto inicial de dados, partindo de condições iniciais idênticas. A partir do conjunto de *n* resultados, o Motf_NVP determinará um conjunto único de resultados.

5.8.1 Responsabilidades

Na aplicação da técnica de programação em n-versões, o Motf_NVP fornecerá mecanismos genéricos que permitam a adoção da técnica independente da funcionalidade das versões participantes, e desempenhará os seguintes papéis:

- *Controlador*: No papel de controlador, ele se torna o responsável pela ativação dos objetos participantes, sincronização das versões, controle de restrições de tempo e coleta dos resultados.

- *Supervisor*: No papel de supervisor, ele supervisiona localmente cada versão, com o objetivo de detectar e tratar ocorrências excepcionais, bem como coletar dados do processo de execução das versões participantes.

- *Árbitro*: No papel de árbitro, ele seleciona o resultado final da execução das n versões e o tratamento das versões faltosas, de acordo com diretivas globais e diretivas fornecidas pela aplicação.

As responsabilidades acima não são centralizadas nesta classe e sim obtidas por associações de herança com outros objetos e serviços prestados por outras classes de objetos, como por exemplo, a classe Motf_Votador.

Cada instância da classe Motf_NVP atua como um meta-objeto associado a um particular objeto tolerante a falhas da aplicação. Nos seus aspectos de funcionalidade e confiabilidade, o Motf_NVP deve atender os seguintes requisitos básicos:

a) Prover, de forma consistente, o conjunto inicial de dados da computações múltiplas.

O conjunto de dados é obtido por interceptação da mensagem dirigida ao objeto tolerante a falhas. O conjunto de argumentos é encapsulado em um objeto[6] que é retransmitido como parâmetro a todos as versões.

b) Determinar um conjunto único de resultados, selecionado dentre o conjunto de resultados fornecidos pelas computações múltiplas.

O resultado da computação de cada versão é fornecido em um objeto[7] e armazenado como um dos componentes do vetor de decisão. Após o término da execução de todas as versões, a votação dos resultados é realizada por um objeto da classe Motf_Votador.

c) Supervisionar e observar os objetos participantes das computações múltiplas. Coletar e fornecer dados obtidos durante a observação do processo de execução.

Todos os objetos participantes são executados[8] por requisição do meta-objeto, que toma o lugar do objeto emissor da mensagem original. O meta-objeto define dois possíveis comportamentos de término de cada uma das versões: normal e excepcional.

A coleta, o registro e o fornecimento de informações a respeito dos objetos participantes é realizado pelo objeto da classe Motf_GrupoTF.

O término normal implica coletar o resultado da execução da versão (para posterior votação) e o término excepcional implica reconfigurar o grupo de objetos participantes.

d)Possibilitar a reconfiguração do grupo de objetos participantes.

---

[6] OpenC++: objeto da classe ArgsPac.

[7] OpenC++: objeto da classe ArgsPac.

[8] OpenC++: por redefinição do método MetaHandleMethodCall.

A reconfiguração do grupo de objetos participantes pode ser feita de duas formas: restaurar a versão por nova instanciação ou abandonar a versão, caso possa ser mantido um grupo mínimo de três versões participantes.

A figura 5.8 mostra o diagrama de interação do meta-objeto da classe Motf_NVP com os objetos participantes da aplicação.

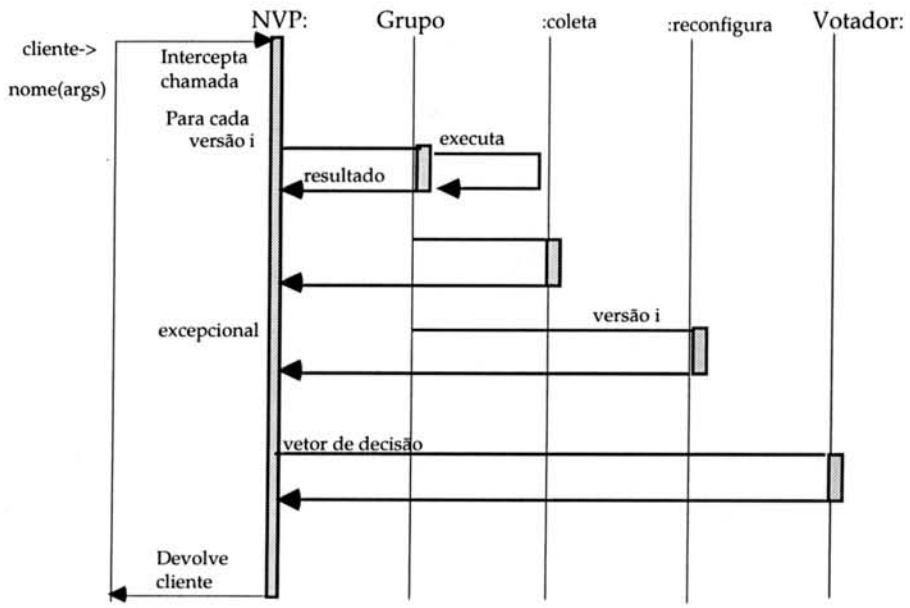


Figura 5.8 - Diagrama de interação

5.8.2 As granularidades

As distintas granularidades (método, objeto ou classe) recebem tratamento homogêneo, ressalvados os cuidados para manter a interface única, representado pelo objeto tolerante a falhas associado ao meta-objeto da classe Motf_NVP.

Métodos diversificados: esta granularidade admite implementações distintas de um método e é tratada da mesma forma que objetos diversificados descendentes de uma mesma classe. Esta abordagem possibilita a execução concorrente de objetos sem utilizar mecanismos de concorrência interna de objetos (disponíveis em poucas linguagens de programação orientadas a objetos).

Objetos diversificados: são descendentes de uma mesma classe ancestral e admitem variações nos dados da classe geradora da instância,

além de implementações distintas do método crítico. É resolvida por instanciações dos objetos participantes a nível de meta-objeto e sua execução é feita através de um único tipo de mensagem. O exemplo anteriormente apresentado na figura 5.5 ilustra várias classes descendentes da classe Interpola, associada à classe Motf_NVP, através da conexão (método reflexivo) InterpolaValor(args).

Classes diversificadas: a exigência é que o método crítico obedeça a mesma interface em todas as classes, sendo responsabilidade do usuário providenciar a uniformidade de interface.

5.9 Classe Motf_RB

De forma semelhante ao Motf_NVP, que apoia a programação em n-versões, O Motf_RB provê um ambiente de execução para a técnica de blocos de recuperação. Os protocolos de ambos os objetos são bastante similares, assim como se assemelham as suas funcionalidades. Desta intersecção, aproveita-se para explorar o mecanismo de herança, na forma de "programação por diferença" [JOH91]. O código e as estruturas de dados comuns são colocados na sua classe ancestral, o Motf_PD - Programação Diversitária. As diferenças essenciais entre essas duas técnicas dizem respeito à forma de execução, teste de aceitação e restauração de estado das alternativas.

5.9.1 Responsabilidades

A forma de execução das alternativas e o teste de aceitação é tradicionalmente representada por uma construção do tipo [RAN75]:

```
ensure <teste de aceitação>
by <alternativa primária>
else by <alternativa secundária>
.....
else by <enésima alternativa>
else erro
```

Na abordagem tradicional, esta construção sintática faz parte do código do programa, juntamente com os demais componentes: um teste de aceitação e as alternativas.

Na arquitetura reflexiva, a alternativa primária é representada por um objeto reflexivo tolerante a falhas e as alternativas e o teste de aceitação são controlados pelo meta-objeto associado. Nesta abordagem, a alternativa primária é mantida inalterada e o custo da utilização de redundância por refelxão computacional resume-se à interceptação da mensagem e a realização do teste de aceitação.

O diagrama da figura 5.9 mostra a dinâmica de execução das alternativas.

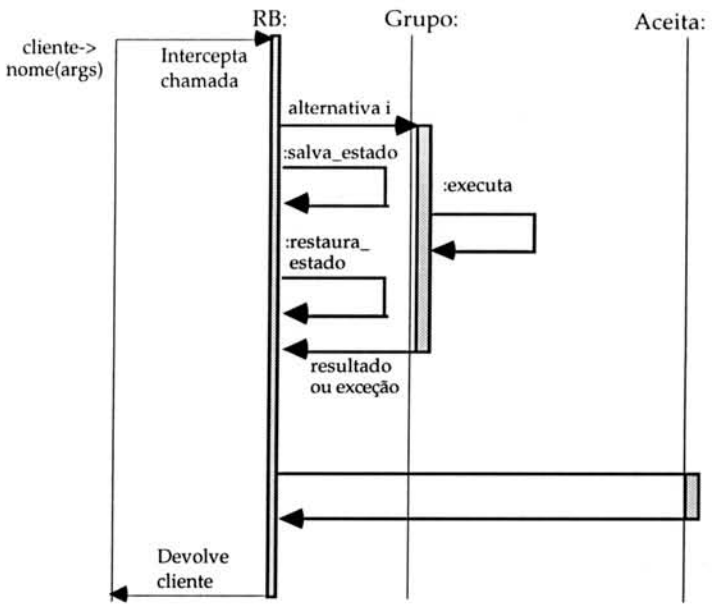


Figura 5.9 - Diagrama de interação de blocos de recuperação

O teste de aceitação é um dos pontos frágeis desssa técnica, visto que resultados corretos podem ser descartados por um teste inadequado. Para reduzir essa fragilidade, o *framework* MOTF permite que o teste de aceitação seja também um objeto reflexivo, controlado por um meta-objeto da classe Motf_RB. Essa definição recursiva objetiva a utilização de alternativas de testes de aceitação, que podem ser derivados de uma mesma classe por polimorfismo dinâmico (i.e., idêntico protocolo, diferentes objetos).

Em caso de falha de uma alternativa, pode existir o ônus da restauração de estado, como abordado a seguir.

5.9.2 Granularidades e preservação de estado

Na técnica de blocos de recuperação, quando o comportamento do objeto *não é funcional* e depende dos valores dos dados da instância referentes à execução anterior para a sua nova ativação (i.e., depende do contexto histórico), torna-se necessário prover o mesmo estado inicial da computação na execução de cada alternativa. Considerando as granularidades (método, objeto, classe), a preservação de estado pode ser feita como segue:

Métodos diversificados: esta granularidade admite implementações distintas de um método, com idênticas estruturas de dados de instância. A alternativa primária é definida no objeto tolerante a falhas da aplicação (método reflexivo) e a alternativa secundária são definidas no meta-objeto associado, não sendo permitidas alterações nos dados de instância do objeto tolerante a falhas.

Objetos diversificados: são descendentes de uma mesma classe ancestral e admitem variações nos dados da classe geradora da instância, além de implementações distintas do método crítico. A arquitetura reflexiva permite controlar variáveis de instâncias, tornando-as reflexivas; cada alteração de uma variável controlada por um meta-objeto é capturada no meta-nível. Para permitir a preservação/restauração do estado, as variáveis da classe do objeto reflexivo tolerante a falhas (classe ancestral) devem ser definidas como reflexivas e mantida esta uniformidade em todas as classes descendentes.

Classes diversificadas: a exigência é que o método crítico obedeça a mesma interface em todas as classes, sendo responsabilidade do usuário providenciar os métodos particulares de preservação/restauração de estado.

5.9.3 Outras técnicas baseadas em diversificação

A classe Motf_Rb também pode ser especializada para a sua utilização em outras técnicas de tolerância a falhas, como a técnica de diversidade de dados. Em [JAL94] encontra-se uma descrição detalhada desta técnica.

No modelo de orientação a objetos, a técnica de diversidade de dados, baseada em re-expressão de dados usando outros tipos ou formas de representação de dados, pode ser reinterpretada por polimorfismo. As

alternativas são constituídas por funções polimórficas que se distinguem por seus tipos de dados.

5.10 Classe Motf_REP

Esta classe abstrata define diferentes serviços de replicação de objetos, baseados no conceito de grupos de objetos tolerantes a falhas. A replicação de objetos tem como requisito a existência de mecanismos de programação concorrente, de forma paralela ou distribuída.

A constituição de grupos de objetos pode ser facilmente obtida por múltiplas instanciações de objetos de uma mesma classe, sendo um objeto a menor unidade de replicação e distribuição. A dificuldade consiste em manter a consistência do estado dos membros participantes em presença de execução concorrente.

A manutenção do estado das réplicas é feita por um protocolo de replicação; cada protocolo corresponde a uma semântica de interação entre as réplicas[9].

A classe Motf_REP define os serviços comuns aos diferentes protocolos:

a) reflexão computacional;

b) constituição e gerenciamento de grupos replicados;

c) preparação dos dados para a execução das versões;

d) salvamento e recuperação do estado dos membros.

Os serviços peculiares a cada protocolo são obtidos por especializações desta classe.

A figura 5.10 esquematiza os principais serviços desta classe, mostrando a sua associação com os objetos da computação. Cada um dos objetos replicado é controlado por um meta-objeto individual, representado por uma sombra, com a responsabilidade de salvamento/restauração do estado do objeto.

---

[9] Ver Capítulo 4, seção 4.7.

Na comunicação com os objetos remotos é utilizado um mecanismo de execução remota. Na arquitetura Motf, os protocolos de distribuição são interpretados por um Mensageiro.

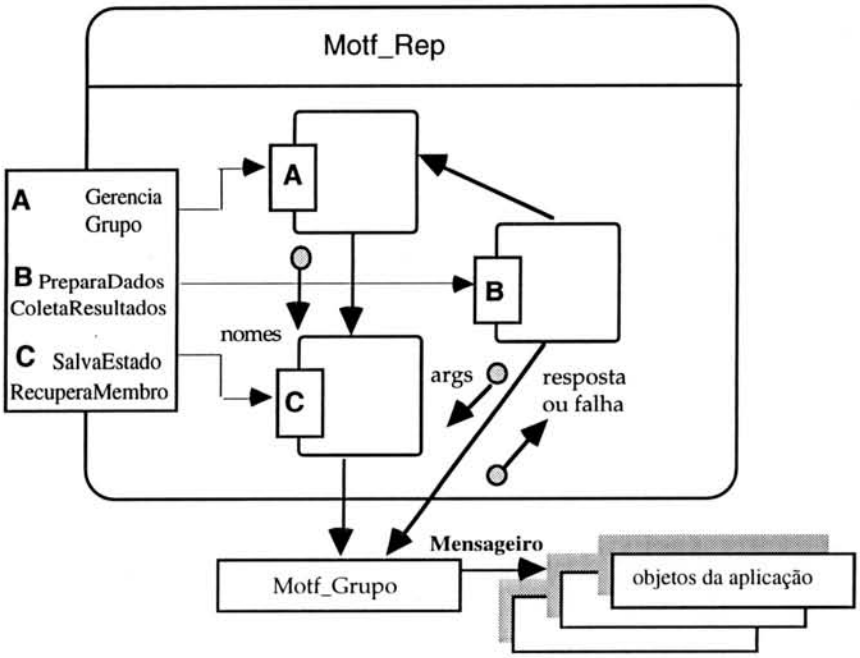


Figura 5.10 - Replicação de objetos

5.10.1 Responsabilidades

Na aplicação, um grupo de objetos replicados é representado por um objeto reflexivo tolerante a falhas. Esse objeto é sempre local, sendo usado para o recebimento de solicitações de serviços a serem realizados pelos objetos replicados. Na arquitetura reflexiva, este objeto é associado a um meta-objeto que fornece um protocolo de replicação.

Os protocolos de replicação correspondem a duas semânticas básicas:

ATIVA: Implementa a semântica de votação. O controle de objetos replicados ativos consiste basicamente em executar concorrentemente todas as n réplicas sobre os mesmos dados em diferentes processadores e determinar um resultado confiável por votação majoritária sobre as $m <= n$ respostas obtidas.

PASSIVA: Implementa a semântica de 'primeira-resposta'. O controle de objetos replicados passivos consiste basicamente em selecionar uma das réplicas para atender ao serviço especificado, sendo as réplicas restantes usadas como alternativas, em caso de falha da réplica primária.

5.10.2 Descrição da classe Motf _REP

CLASS **Motf _REP** IS ObjRefl
DESCRIPTION Esta classe abstrata define diferentes serviços de replicação de objetos, baseados no conceito de grupos de objetos tolerantes a falhas.
PROVIDED_INTERFACE [-- Descrição dos recursos fornecidos a outros objetos.]
Gerencia_Grupo
DESCRIPTION É decomposta em um conjunto de outras operações internas. Permite a criação, reconfiguração e destruição de grupos de objetos replicados.
Prepara_Dados
DESCRIPTION Normaliza os argumentos de execução dos objetos do grupo.
Coleta_Resultados
DESCRIPTION Recebe os resultados da execução dos membros do grupo.
virtual SalvaEstado
DESCRIPTION Faz uma cópia do estado do objeto.
virtual RecuperaMembro
DESCRIPTION Restaura o estado do objeto.
EXCEPTIONS [-- Exceções que podem ser transmitidas a outros objetos.]
Falha_REP
DESCRIPTION Indica que não foi possível fornecer uma resposta ao cliente do objeto tolerante a falhas. Pode ser decomposta para explicitar o tipo de falha: sem aceitação de resultados, falhas na restauração de estado, etc.
REQUIRED_INTERFACE [-- Descrição dos recursos de outros objetos.]
CLASS [-- Lista de outras classes Motf usadas]
Motf_Grupo
ENVIRONMENT_CLASS [-- Lista de outras classes usadas]
Mensageiro
END_CLASS **Motf_REP**

Observações: No modelo de reflexão computacional de OpenC++, as variáveis de instância podem ser declaradas como reflexivas, permitindo operações de salvamento e recuperação de estado. Quando as réplicas são puramente funcionais, isto é, seu estado não depende de nenhuma execução anterior, a recuperação de membros pode ser feita por nova instanciação e com cópia de objetos (operação **Clone** de Motfs.)

5.10.3 Descrição da classe Motf_RA

CLASS **Motf_RA** IS **Motf_REP**
DESCRIPTION Esta classe define protocolos de replicação para objetos distribuídos que busquem assegurar a manutenção de um estado consistente entre as réplicas e assegurar a disponibilidade de um número mínimo de réplicas. Objetos desta classe são instanciados como meta-objetos de objetos da aplicação.
PROVIDED_INTERFACE [-- Descrição dos recursos fornecidos a outros objetos.]
OPERATIONS [-- Métodos públicos definidos pela classe]

EXCEPTIONS [-- Exceções que podem ser transmitidas a outros objetos.]

END_CLASS **Motf_RA**

Observações:

5.10.4 Descrição da classe Motf_RP

CLASS **Motf_RP** IS **Motf_REP**
DESCRIPTION Implementa a semântica de 'primeira-resposta'.Objetos desta classe são instanciados como meta-objetos de objetos da aplicação.
PROVIDED_INTERFACE [-- Descrição dos recursos fornecidos a outros objetos.]
TYPES [-- Tipos públicos definidos pela classe.]
CONSTANTS [-- Constantes públicas definidas pela classe.]
OPERATIONS [-- Métodos públicos definidos pela classe]
EXCEPTIONS [-- Exceções que podem ser transmitidas a outros objetos.]
REQUIRED_INTERFACE [-- Descrição dos recursos de outros objetos.]
CLASS [-- Lista de outras classes Motf usadas]
ENVIRONMENT_CLASS [-- Lista de outras classes usadas]
TYPES [-- Lista de nome_da_classe. nome_do_tipo usado]
OPERATIONS [-- Lista de métodos usados]
EXCEPTIONS [-- Lista de exceções capturadas]
END_CLASS **Motf_RP**

Observações:

5.11 Classe Votador

Dois aspectos devem ser considerados para a implementação de algoritmos de votação para selecionar um único resultado dentre os resultados fornecidos pelos objetos participantes.

Primeiro, o mecanismo de votação necessita coletar os resultados disponíveis da execução das versões, colocá-los em uma estrutura de dados na forma de um vetor de decisão e determinar o resultado da votação[KEL91]. Para que o mecanismo de votação possa definir uma relação de equivalência, ele deve comparar os resultados gerados pelas versões. Como estes resultados possuem um tipo de dado a eles associado, o votador depende do tipo de dado e de uma relação de equivalência, como exemplificado abaixo.

Tipo	Relação de equivalência
Inteiro	Identidade
Booleano	Identidade
Ponto flutuante	Similaridade[10]
Caractere	Identidade[11]
String	Similaridade[12]

[10] Identificar uma relação de equivalência baseada em precisão.

[11] Se for usado o mesmo código de representação.

[12] Pode admitir variações 'cosméticas', como espaços.

Segundo, diversos algoritmos podem ser utilizados para a determinação do resultado único, tais como: maioria absoluta, maioria relativa, média ou mediana dos valores, entre outros.

Os diversos tipos de dados que podem constituir o vetor de decisão são modelados por meio de uma classe parametrizada (genérica), como esquematizada na figura 5.11.

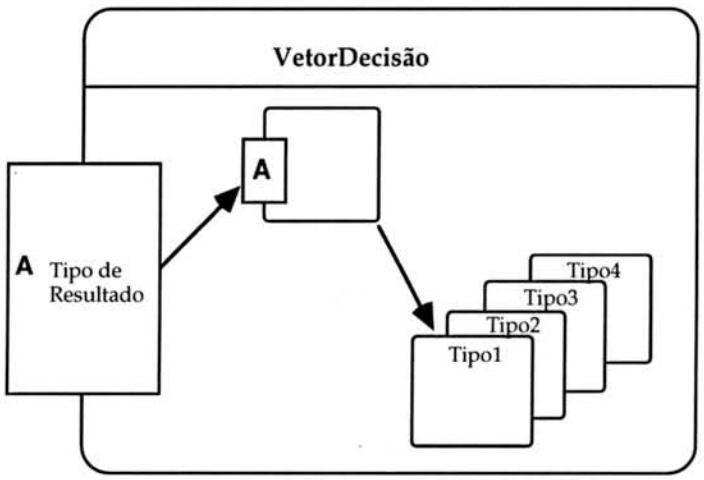


Figura 5.11: Classes genéricas de vetor de decisão

Enquanto as classes genéricas permitem variações nos tipos de dados, vários mecanismos de decisão polimórficos podem implementar distintos comportamentos usando o mesmo nome. Eles podem diferir nos argumentos ou podem possuir idênticos protocolos. Neste último caso existirá a herança dinâmica.

Os mecanismos de decisão são reunidos em uma classe, denominada Motf_Votador (figura 5.12), cujos métodos recebem como argumento um objeto da classe VetorDecisão e devolvem um único resultado (ou sinalizam erro) do mesmo tipo dos elementos do vetor de decisão.

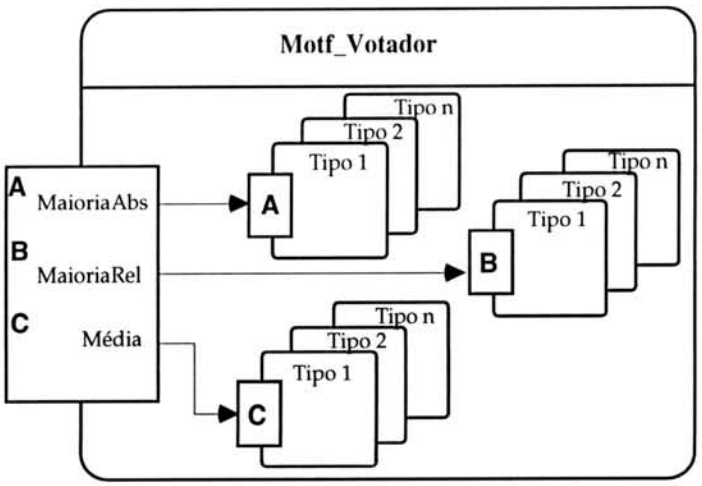


Figura 5.12 : Métodos polimórficos de votação

A aplicação poderá utilizar um dos algoritmos de votação polimórficos fornecidos ou poderá definir o seu próprio algoritmo de votação. O objeto definido pela aplicação, para atuar como votador, poderá herdar desta classe todos os métodos e estruturas de dados que lhe interessem, tais como métodos de comparação, seleção por maioria absoluta, seleção por maioria relativa, entre outros, particularizando apenas as diferenças.

5.12 Comentários finais

A modelagem das classes de objetos *unifica as estratégias* de tolerância a falhas a partir de dois conceitos básicos: o conceito de objeto reflexivo tolerante a falhas (nível da aplicaçãoD_0) e o conceito de grupos de objetos controlados por um meta-objeto (meta-nível D_1) associado ao objeto reflexivo tolerante a falhas. A classe representante de um objeto reflexivo tolerante a falhas é usada como ancestral de todas as classes que gerenciam a redundância de objetos. Os objetos redundantes, sejam replicados ou diversificados, são organizados em um grupo através da classe Motf-GrupoTF, que organiza uma interface comum a todos os participantes.

Dentre as classes descritas neste capítulo, algumas são abstratas e outras são concretas. As classes abstratas são projetadas para representar comportamentos parciais de objetos, a serem complementados com outros comportamentos em classes descendentes, e/ou para padronizar uma interface que poderá ser concretizada por mais de uma subclasse, cada qual

contendo uma representação distinta. Um exemplo é a classe que implementa protocolos de execução de objetos distribuídos.

As classes concretas, organizadas em uma biblioteca de classes, podem ser usadas de duas maneiras: (a) para instanciar objetos, mantendo assim a sua definição original e, (b) para derivar novas classes, modificando ou complementando a sua definição original.

Capítulo 6

CONSIDERAÇÕES FINAIS E CONCLUSÕES

Reúne e sintetiza as observações, comentários e conclusões constantes nos capítulos anteriores. Aponta perspectivas de continuação da pesquisa.

6.1 O desenvolvimento da pesquisa

Esta pesquisa foi iniciada pelo estudo de técnicas de tolerância a falhas em diversos modelos de linguagens de programação (imperativo, paralelo, distribuído, lógico, funcional e orientado a objetos), buscando, em cada modelo, identificar os padrões conceituais e as facilidades oferecidas para o desenvolvimento de aplicações tolerantes a falhas. Emergiram desse estudo as primeiras conclusões [LIS94], resumidas e comentadas nas seções 6.2 e 6.3.

O desenvolvimento da pesquisa, centrado na questão da simplificação do processo de desenvolvimento de aplicações tolerantes a falhas, amparou-se em três componentes: Tolerância a Falhas, Orientação a Objetos e Reflexão Computacional.

O primeiro componente define o foco de interesse, o segundo define a abordagem estrutural e o terceiro identifica a forma de implementação. Os dois últimos componentes são focalizados separadamente nas seções 6.4 e 6.5. A seção 6.6 reúne os três componentes, na abordagem dada neste trabalho e a seção 6.7 apresenta as conclusões finais e sugere perspectivas de continuidade.

6.2. Considerações sobre técnicas de tolerância a falhas

As técnicas clássicas de tolerância a falhas, programação em n-versões e blocos de recuperação, inspiram-se em métodos usuais de prevenção de catástrofes: a existência e a disponibilidade de alternativas e a multiplicação de recursos.

Em decorrência disto, o custo de desenvolvimento de software tolerante a falhas é muito alto, inviabilizando a sua plena utilização e tornando estas técnicas de utilidade questionável. Em contrapartida, a única forma conhecida de tolerar falhas de projeto e programação, na fase operacional do software, é a diversidade de projetos e de implementação. A replicação de componentes e a técnica de diversidade de dados são menos onerosas em termos de custo de desenvolvimento, porém têm um espectro de aplicabilidade mais restrito.

Além da sua aplicação na construção de software tolerante a falhas, outros usos também são sugeridos para as técnicas de programação diversitária: testes de componentes e transição de versões.

A técnica de programação em n-versões encontra aplicação na área de testes [AVI85][KEL91]. A existência de diferentes versões de similar funcionalidade podem ser usadas para testes comparativos[1] com geração de um grande número de casos de testes sem ser necessário determinar antecipadamente os valores corretos das respostas. Considera-se como corretos os resultados coincidentes entre as versões; quando são obtidos resultados divergentes, considera-se que alguma das versões apresenta problemas.

Esta técnica tardia de depuração, pode ser usada com diversos objetivos: como técnica suplementar de testes [ABB90], como técnica para viabilizar testes de sistemas distribuídos que se caracterizam por uma grande diversidade de estados da computação resultantes de combinações de componentes concorrentes [KEL91], ou ainda, para selecionar componentes para uma aplicação. Entre os componentes testados, pode-se selecionar o mais adequado à aplicação.

O *framework* MOTF, ao separar as funcionalidades da aplicação de seus requisitos não-funcionais, facilita a introdução e retirada dos objetos redundantes e os meta-objetos que fornecem os serviços de tolerância a falhas. Os objetos da aplicação mantém as suas classes e interfaces originais, permitindo a reconfiguração da aplicação, para atender finalidades como: mudar a técnica de tolerância a falhas, substituir, aumentar ou reduzir os componentes redundantes, ou mesmo retirar as características de tolerância a falhas usadas com propósito de depuração ou seleção de componentes da aplicação. Exemplos apresentados no capítulo 4 ilustram essas facilidades.

6.3 Considerações sobre modelos de linguagens de programação

As linguagens de programação imperativas sempre foram o paradigma do desenvolvimento de todas as técnicas, métodos, ferramentas e objeto de experimentos de tolerância a falhas e, mais especificamente. de mecanismos de tratamento de exceções, com base nos quais algumas técnicas são implementadas. A norma MOD00-55 [PLA93], de 1991, do Ministério da Defesa do Reino Unido sugere fortemente que sejam usadas linguagens orientadas a procedimentos no desenvolvimento de sistemas de segurança crítica, ao invés de linguagens *assembler* ou outras linguagens "não-convencionais".

---

[1] *back-to-back test.*

Todavia, a literatura na área de tolerância a falhas sugere outras abordagens: linguagens de programação em lógica e linguagens funcionais.

Abbot [ABB90] advoga a idéia de que uma linguagem de programação em lógica é ideal para a implementação de sistemas tolerantes a falhas, amparado nas seguintes razões básicas: mecanismo de retrocesso que inclui, automaticamente, o salvamento/restauração de variáveis, que são características básicas da programação diversitária, especificações executáveis, que permitem ao programador trabalhar ao nível de especificação e mecanismos de implementação de programação diversitária de forma relativamente simples.

A par destas facilidades, restrições podem ser apontadas, nas linguagens de programação em lógica, que limitam severamente a sua irrestrita aplicação em sistemas tolerantes a falhas. Desempenho, implementação segura de componentes encapsulados que possibilitem o confinamento de danos decorrentes de falhas, mecanismos poderosos de tratamento de exceções e estruturas de dados robustas, implementadas através de tipos abstratos de dados, não encontram apoio nas linguagens de programação em lógica. Também o aspecto de controle de tempo em sistemas críticos de tempo real não encontra facilidades diretas para a sua utilização.

O modelo de programação funcional também encontra adeptos: Jagannathan e Ashcroft [JAG91] sugerem que linguagens funcionais paralelas que implementam o modelo computacional baseado em demanda são naturalmente adequadas para a utilização de redundância temporal e espacial. No modelo de demanda, uma operação é executada se e quando seu resultado é exigido e seus operandos estão disponíveis. A redundância espacial (computação simultânea) seria obtida por demandas redundantes dos operandos e seus resultados submetidos a um votador. A redundância temporal (repetir a computação após um insucesso) seria obtida recalculando o valor omisso.

De forma semelhante ao paradigma da programação em lógica, as linguagens de programação funcionais encontram maior aplicabilidade na prevenção de falhas do que no desenvolvimento de aplicações tolerantes a falhas. Aplicações críticas que, por construção, devem envolver redundância de componentes críticos e algoritmos excepcionais encontrariam alguns pontos de apoio neste paradigma: funções críticas podem ser formalmente especificadas e validadas, usando sempre a mesma

ferramenta - a própria linguagem de programação funcional. O forte embasamento matemático das linguagens funcionais predispõe o programador a uma disciplina de programação bastante severa; esta habilidade pode ser explorada para a construção de alternativas de blocos de recuperação ou programação em n-versões, contribuindo para a independência das versões.

No *framework* MOTF a linguagem de prototipação escolhida foi C++, pelas razões apontadas no capítulo 2, seção 2.2. Esta linguagem convive com outras linguagens de programação: componentes de um programa podem ter sido implementados em outras linguagens. Porém, se impõe a restrição do modelo da aplicação tolerante a falhas - cada componente crítico é um objeto - e da técnica de implementação - a reflexão computacional.

6.4 Considerações sobre o modelo de orientação a objetos

A individualidade do modelo tem como característica mais marcante o mecanismo de herança. Sob o ponto de vista de desenvolvimento de programas, este mecanismo promove uma estruturação de programas completamente distinta de outros modelos, seguindo o princípio destacado na própria denominação do modelo: orientação a objetos.

O comportamento tolerante a falhas pode ser especificado e confinado a um nível de granularidade menor e o confinamento de dados e operações permite o isolamento de componentes críticos. A modelagem do comportamento de objetos por meio de classes abstratas pode ser concretizada através de implementações distintas, separando nitidamente os aspectos comportamentais e estruturais.

Objetos podem ser dinamicamente criados e multiplicados, sempre obedecendo a estrutura e o comportamento detalhado em sua classe; a construção de programas tolerantes a falhas pode se beneficiar deste dinamismo para a replicação e recuperação de componentes.

O modelo de orientação a objetos também proporciona a estrutura de implementação de reflexão computacional. Mensagens podem ser reinterpretadas ou redirecionadas para a realização de atividades que possam complementar ou mesmo modificar o comportamento especificado para o objeto destinatário da mensagem.

A adequação do modelo, explicitada ao longo deste trabalho, também já foi percebida por diversos pesquisadores, que tem abordado vários aspectos de tolerância a falhas no modelo de orientação a objetos [SHR95], [SHR94], [RUB94a], [RUB94b],[XU95], [FAB95].

As considerações sobre a adequação do modelo de orientação a objetos para o desenvolvimento de aplicações tolerantes a falhas são resultantes de pesquisas e não de experimentação controlada. Os experimentos sobre programação diversitária mais recentemente descritos na literatura [BIS88],[KEL88],[KEL91] foram conduzidos sobre o modelo de programação imperativa, usando as linguagens FORTRAN e PASCAL.

O modelo de orientação a objetos vai além de mecanismos de linguagens de programação. Ele introduz novas técnicas de modelagem, análise, projeto e estruturas de programas; sua contribuição para o efetivo aumento de confiabilidade de software ainda não foi demonstrado através de experimentos controlados.

6.5 Considerações sobre reflexão computacional

Linguagens de programação, por concepção, possuem uma semântica bem definida, a ser obedecida pelos seus usuários no desenvolvimento de aplicações. A reflexão computacional muda esta idéia, ao permitir alterações na implementação da linguagem. Por exemplo, a alteração da classe de um objeto flexibiliza o conceito de que objetos herdam estaticamente a estrutura de sua classe ancestral. Ou ainda, ao interceptar uma mensagem dirigida a um objeto e reenviá-la ao seu meta-objeto, flexibiliza o conceito de que o objeto, ao receber uma determinada mensagem tem o comportamento ditado pelo método ativado por tal mensagem.

Esta flexibilização tem sido implementada em linguagens de programação através de um *protocolo de meta-ojetos* ('MOP - metaobject protocol')[KIC91], que permite operações com meta-objetos.

Um aspecto importante da reflexão computacional é a possibilidade de monitoração do comportamento de objetos de nível base e de interferir na computação, substituindo ou aumentando as funcionalidades desses objetos, de forma transparente.

O *framework* MOTF explora essa possibilidade, utilizando meta-objetos para adicionar a objetos selecionados da aplicação,

comportamentos referentes a requisitos não-funcionais da aplicação, sem interferir na funcionalidade básica esperada pelos clientes desses objetos.

6.6 Arquitetura reflexiva para a tolerância a falhas no modelo de orientação a objetos.

Como mencionado no Capítulo 4 - Seção 4.5, a arquitetura reflexiva do *framework* MOTF para a implementação de aplicações tolerantes a falhas é organizada em três níveis: o nível base D_0 que corresponde aos objetos da aplicação, o meta-nível D_1 que corresponde aos meta-objetos que controlam os objetos de nível base e que implementam os serviços de tolerância a falhas, e o meta-meta-nível D_2, onde se encontram os mecanismos que dão suporte aos meta-objetos. Por exemplo, mecanismos para controle de concorrência podem ser usados no nível D_1, porém são implementados em D_2.

Os diversos níveis objetivam a reutilização e a transparência dos serviços. Porém, a transparência total de todos os serviços não é possível. em parte devido à formulação independente do domínio da aplicação, que implicam particularizar porções de código e também devido à exigência de algumas técnicas de tolerância a falhas. Nas técnicas que exigem diversificação de componentes, não é possível prescindir do fornecimento, pelo programador, das diferentes versões ou alternativas e de algoritmos particulares de aceitação.

A utilização de um nível de serviços para atender as funcionalidades da aplicação e de outro nível para (tentar) assegurar que estas funcionalidades sejam atendidas de forma transparente ao usuário dos serviços da aplicação, distingue claramente os dois tipos de atividades e propicia diversas formas de reutilização de componentes. Tendo em mente a complexidade intrínseca de software tolerante a falhas, a reutilização de componentes torna-se bastante atraente, pois, além de liberar o programador de aplicação de desenvolver código estranho ao domínio da aplicação, tende a reduzir a incidência de falhas na aplicação.

6.7 Conclusões finais e perspectivas

Neste trabalho pretendeu-se demonstrar, por indução, a proposição geral da adequação da arquitetura reflexiva no modelo de orientação a objetos objetivando a simplificação do processo de desenvolvimento de software tolerante a falhas. Com base em

conhecimentos adquiridos a partir de experimentos de menor generalidade [LIS95a], [LIS95a], [CLE95], [ANC95] foram criadas abstrações dos componentes de um *framework* genérico.

No decorrer da maturação das idéias aqui propostas, trabalhos correlatos foram sendo divulgados [RUB95a], [RUB95b], [FAB95], [XU94] corroborando e complementando essas idéias.

Um contribuição deste trabalho que o distingue e merece destaque é a abordagem de tolerância a falhas centrada no conceito de um objeto tolerante a falhas que representa um objeto da aplicação cujo serviço é considerado crítico, sendo este serviço fornecido por um grupo de objetos redundantes. O grupo de objetos redundantes interage semânticamente por votação ou por 'primeira-resposta', podendo ser implementado sob a forma de objetos replicados ou objetos diversificados. Estes últimos podem ser diversificados em diferentes granularidades.

A modelagem das classes de objetos *unifica as estratégias* de tolerância a falhas a partir de dois conceitos básicos: o conceito de objeto reflexivo tolerante a falhas (nível D_0) e o conceito de grupos de objetos controlados por um meta-objeto (nível D_1) associado ao objeto reflexivo tolerante a falhas.

Sugere-se que a implementação do *framework* MOTF seja feita em duas etapas, que correspondem às Fases I e II descritas no capítulo 4.

A configuração de aplicações tolerantes a falhas (Fase I) exige interação com o programador da aplicação para a obtenção de diretivas, visto que a sua missão é oferecer serviços de tolerância a falhas e buscar informações para a configuração. Estas informações incluem os objetos que devem ser tornados tolerantes a falhas, os demais componentes redundantes e a estratégia de tolerância a falhas a ser adotada.

No contexto deste trabalho, as diretivas são sugeridas sob forma de comentários no texto fonte da aplicação, servindo como base para a expansão de código através de um pré-processador. Alternativas mais sofisticadas podem ser estudadas: interfaces gráficas e ambientes inteligentes de desenvolvimento.

Estes últimos podem não apenas auxiliar na configuração da aplicação tolerante a falhas, mas também contribuir para a prevenção de falhas dos objetos da aplicação. Sob o enfoque de prevenção de falhas, ambientes inteligentes de desenvolvimento de software podem contribuir para:

- forçar a utilização adequada de mecanismos de tratamento de exceções que contribuam para a obtenção de objetos mais robustos;
- minimizar a incidência de erros de interface dos objetos, criticando e assistindo ao programador nesta tarefa;
- minimizar os erros funcionais de objetos, com a criação e auxílio na utilização de classes/métodos já testados e verificados.

A notação utilizada para a modelagem das classes MOTF foi baseada no método HOOD - Hierarchical Object-oriented Design, que abrange as diversas fases de projeto. Nesta notação, um objeto é descrito por um Object Description Skeleton - ODS e objetos ditos terminais são descritos com um nível detalhamento que permite a sua direta implementação em código Ada. Para a geração de código fonte em Ada, na forma de protótipos, o método HOOD possui uma equivalência entre o ODS e construções da linguagem Ada, que é uma linguagem baseada em objetos, carecendo de características fortes de linguagens orientadas a objetos, como herança e polimorfismo.

Uma vez que a plataforma proposta para a implementação dos objetos MOTF- FASE II baseia-se em C++/OpenC++ foi feita uma adaptação da notação para acomodar as diferenças, mantendo-se as equivalências. Esta adaptação não contempla o nível de detalhamento suficiente para a geração automática de código fonte em C++ ou mesmo para a construção de ferramentas para a verificação da consistência do projeto das classes. Este aspecto pode ser objeto de mais estudos, interessantes pela dupla abordagem: a abordagem formal - especificação da notação - e a abordagem de prototipação - geração de código. Ambas podem servir de base a trabalhos futuros, genéricos, não sendo específicas do *framework* MOTF.

Como sugestão final, impõe-se a experimentação do modelo de orientação a objetos para averiguação de suas contribuições para a prevenção, detecção e tolerância a falhas, ao longo do processo de desenvolvimento de software com alta exigência de confiabilidade.

ANEXO 1

Notação

1 Introdução

Na seleção de um método de modelagem para a representação dos MOTF, foram estabelecidos os seguintes critérios:

a) Capturar os aspectos funcionais mais importantes do domínio de tolerância a falhas;

b) Descrever de várias formas os objetos do *framework* MOTF, apresentando diferentes perspectivas, como:

b.1) Descrever a estrutura dos objetos componentes: seus atributos, suas operações e suas interfaces;

b.2) Permitir a visualização dos aspectos comportamentais dos objetos, seu relacionamento com outros objetos e com seu meio-ambiente operacional;

b.3) Descrever a sua hierarquia de composição e sua hierarquia de uso;

c) Assegurar a consistência da especificação.

Examinando a lista acima, pode-se deduzir a adequação da utilização de um método não formal de especificação, que utilize representação gráfica dos objetos e seus relacionamentos, mas que não descarte a possibilidade de utilização de especificação formal no projeto detalhado do objeto.

2 O método HOOD

O método HOOD - Hierarchical Object Oriented Design foi desenvolvido em 1987 por um consórcio de CISI Ingenierie, CRI A/S e Matra Espace, contratado pela ESA - European Spacial Agency, para ser usado como um método de projeto para ADA, com a intenção de utilizar prioritariamente esta linguagem para implementação de sistemas aeroespaciais embutidos. HOOD tem sido utilizado em projetos de grandes sistemas aeroespaciais, como Columbus, Ariane V e European Fighter Aircraft [ROB92].

Este método é derivado de OOD - Object Oriented Design [BOO94], adicionando formalidade e hierarquias de decomposição e de uso e destina-se a apoiar as seguintes fases do desenvolvimento: definição do problema,

elaboração de uma estratégia informal de solução, formalização da estratégia e formalização da solução.

Enquanto que nas fases iniciais do processo de desenvolvimento a preocupação é uma clara compreensão do problema, e a determinação de seus requisitos funcionais e não funcionais deforma a permitir a elaboração de um esboço preliminar de solução, a fase de *formalização da estratégia* encaminha o processo de modelagem. São então capturados os conceitos fundamentais da estratégia de solução, identificados os objetos e suas operações, determinada a composição de objetos através de suas interações, culminando com a representação gráfica dos objetos, discriminando a sua decomposição hierárquica e a estrutura de uso.

A formalização da solução é voltada a cada objeto: a descrição formal de suas interfaces, a descrição formal de sua estrutura de controle e seu comportamento semântico.

3 Aspectos relevantes de HOOD

Salientam-se no método HOOD [HOO89] os seguintes aspectos relevantes para a especificação de MOTFs:

- *Aspectos comportamentais*: HOOD faz uma clara distinção entre objetos *passivos*, que não possuem qualquer semântica relativa a controle, e objetos *ativos*, que reagem a estímulos e respondem de acordo com seu estado interno e o tipo de requisição (síncrona, assíncrona ou temporal).

- *Fluxo de controle*: além do fluxo normal de execução, representado pelo relacionamento de uso entre os objetos, HOOD possui uma representação explícita para o fluxo de controle *excepcional*, que pode ocorrer na forma de um retorno anormal do fluxo de controle durante a execução de uma operação.

- *Meio-ambiente*: objetos a serem usados pelo sistema mas que não fazem parte de seu projeto, podem ter as suas interfaces especificadas por meio de um objeto-ambiente, sem serem incorporados à hierarquia do projeto. Este aspecto é particularmente interessante para a agregação de objetos de outros domínios ou a reutilização de objetos.

- *Objetos virtuais*: sistemas distribuídos podem ser projetados sob a visão de uma rede de nodos virtuais que se comunicam. A representação HOOD de sistemas distribuídos é feita através de Objetos de Nodos Virtuais, similares a objetos ativos. que se comunicam através de interfaces bem definidas e de acordo com protocolos de transmissão pré-estabelecidos.

- *Formalismo textual*: a descrição detalhada de objetos é feita por uma linguagem HOOD-PDL, que possui uma sintaxe formal adequada à utilização de ferramentas automáticas de validação.

- *Regras*: regras formais são definidas em HOOD, permitindo a verificação da consistência e da completeza da descrição do projeto. Estas regras são aplicáveis a cada passo do processo de decomposição do objeto.

4 Diagramas

Diagramas são usados para a visualização de objetos e classes de objetos. Um diagrama HOOD, como o apresentado na figura 1, esquematiza as propriedades estáticas de um objeto (ou classes de objetos), visíveis externamente.

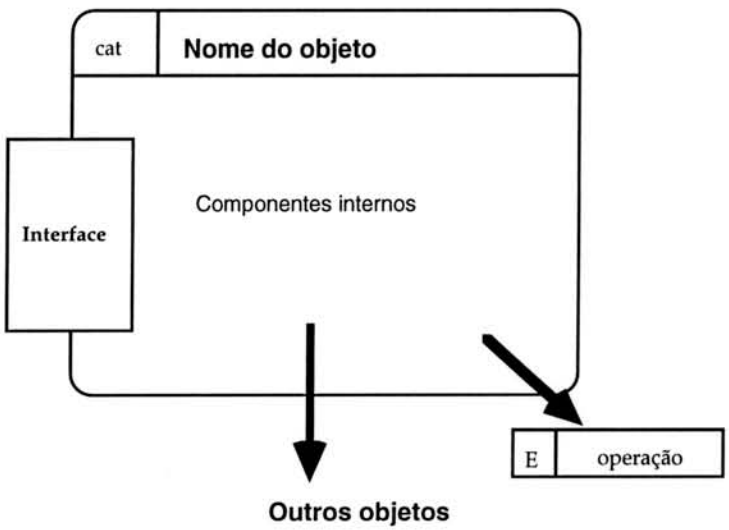


Figura 1 - Diagrama de um objeto

A **Interface** define o formato das mensagens que o objeto pode receber, isto é, as solicitações de execução de suas operações. Os **componentes internos**

implementam as operações descritas na **Interface**, usando dados, operações e objetos internos. No canto superior esquerdo do diagrama, uma caixa (**cat**) é usada para indicar a categoria do objeto: passivo (sem marcação), ativo (letra **A** - Active) ou distribuído (letra **V** - Virtual).

Aspectos de interação entre objetos são indicados na figura 1 por setas que atravessam as bordas do diagrama. Um objeto A usa um objeto B, quando A necessita uma ou mais operações fornecidas pelo objeto B (detalhado em outro diagrama).

Um caso especial refere-se ao objeto denominado *Environment Objetc*, cujo diagrama é identificado pela letra E. Ele representa um objeto que fornece operações ao objeto representado no diagrama, porém não faz parte do projeto do sistema: pode ser um objeto já existente e que está sendo reutilizado ou representar uma ou mais operações fornecidas pelo ambiente de suporte. Esse objeto não é único: pode representar diversos objetos e cada diagrama pode possuir sua própria lista de *Environment Objects*.

No contexto deste trabalho, o diagrama apresentado na figura 1 é usado para a representação gráfica de **classes** de objetos, e não suas instâncias, com o objetivo de dar um cunho mais genérico ao projeto e evitar ambiguidades em relação à classes parametrizadas que não podem ser diretamente instanciadas. O método HOOD prevê a representação de classes de objetos com múltiplas instâncias, mas com a interpretação de classes genéricas da linguagem ADA[1]: a classe deve possuir parâmetros de tipos ou dados, sendo suas classes particularizadas por esses parâmetros, para então permitir instanciação.

Classes de objetos podem possuir diversos relacionamentos estáticos (herança) e dinâmicos (uso). A representação do relacionamento de herança de classes Motf é mostrada por uma árvore de hierarquia, como ilustrado na figura 2.

---

[1] ADA 83: baseada em objetos

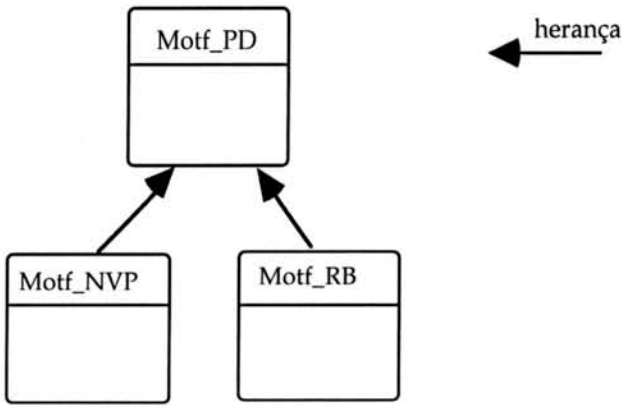


Figura 2 - Hierarquia de classes

OBS: No método HOOD, o projeto é baseado no princípio de decomposição sucessiva, onde cada nível de decomposição representa um relacionamento "pai-filho". Um objeto "pai" é decomposto em um conjunto de objetos "filhos", os quais, coletivamente fornecem a mesma funcionalidade do ancestral. A decomposição encerra quando os objetos não são mais passíveis de decomposição. Este relacionamento é típico de decomposição por *agregação*, sendo representado neste método como objetos *internos* ao objeto ancestral, associados por setas pontilhadas, como esquematizado de forma simplificada na figura 3.

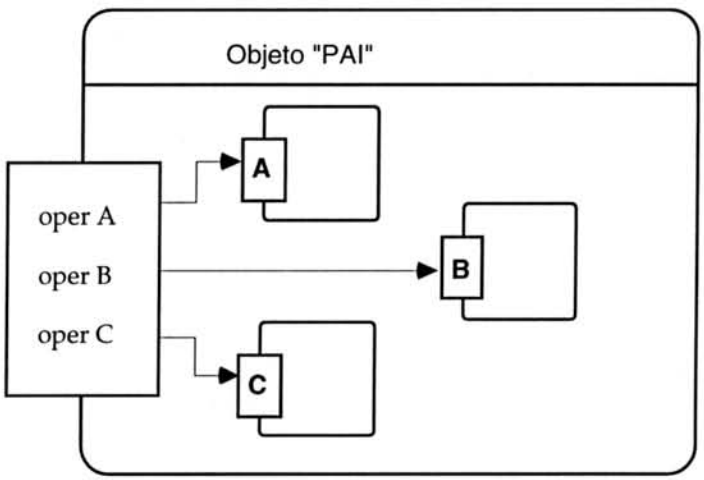


Figura 3 - Decomposição de objetos

O relacionamento de 'uso' é representado em HOOD por setas de linhas cheias, admitindo mesclar as associações em um mesmo diagrama, como esquematizado na figura 4.

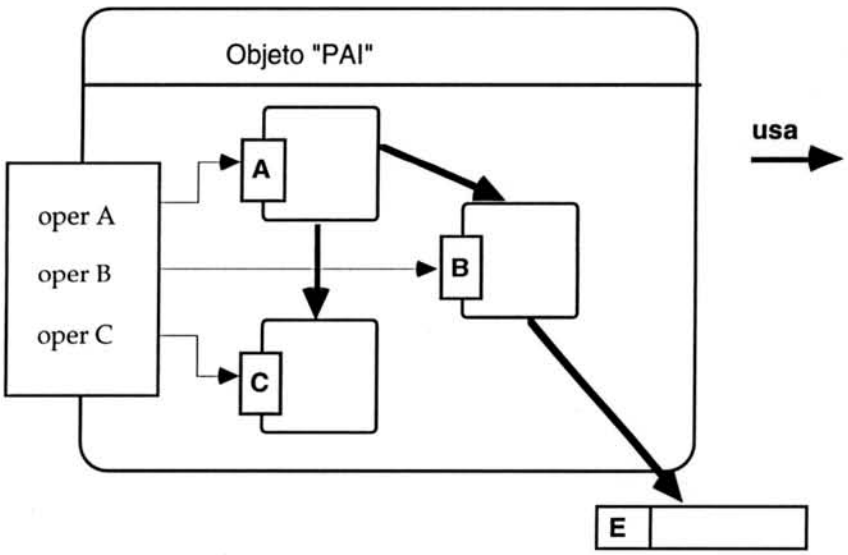


Figura 4 - Decomposição de operações

Na figura 4, a operação A é implementada por um objeto "filho", que, por sua vez, utiliza operações de outro objeto "filho".

Além da semântica estática de objetos que representa a organização da estrutura do programa, a semântica dinâmica também dever ser representada na fase de projeto. No processo de execução, objetos interagem com outros objetos e assim determinam a organização dinâmica do programa, ou seja, o fluxo de controle de execução.

Os aspectos dinâmicos podem ser representados através de diversas técnicas. Na descrição da interação de objetos e meta-objetos, a representação utilizada é o diagrama de interação. De acordo com Booch, um diagrama de interação é usado para representar um cenário de execução, e se constitui numa generalização dos diagramas de Rumbaugh e Jackson [BOO94].

Um diagrama de interação mostra na parte superior a lista de objetos que interagem no decorrer do processo de execução e na parte lateral os eventos, operações e mensagens relacionadas com os objetos. Um diagrama de interação está esquematizado na figura 5, mostrando o cenário de execução seqüencial , onde o objeto NVP envia a mensagem InterpolaValor(args) para os

objetos ver1, ver2 e ver3, e a mensagem MaioriaAbs(args) para o objeto Votador. O tempo relativo de execução é representado verticalmente, de cima para baixo, ordenando assim a seqüência das mensagens.

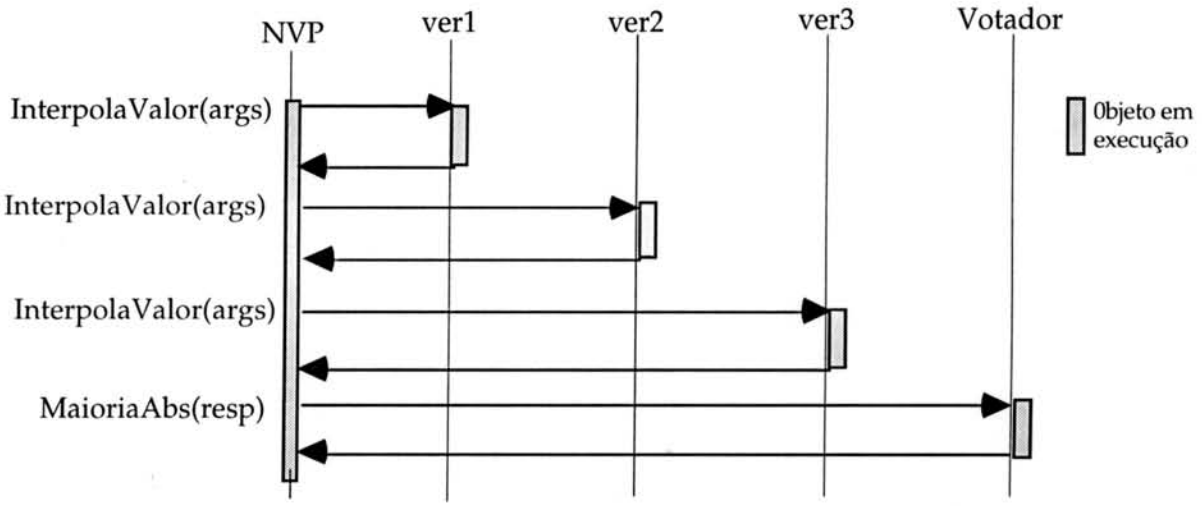


Figura 5 - Diagrama de interação

5 Descrição formal de objetos

A descrição de objetos em HOOD é feita através de um Object Description Skeleton - ODS, que é uma descrição formal derivada a partir do diagrama, e que pode ser verificada quanto aos aspectos de completude, consistência interna e coerência de interfaces. A notação HOOD/ODS é definida como segue:

```
ods:: =        OBJECT object_name IS object_type
                  object_description
                  implementation_constraints
                  provided_interface
                  required_interface
                  data flows
                  opcs
               END_OBJECT object_name
```

sendo, de forma simplificada:

```
object_type ::= <PASSIVE | ACTIVE | ENVIRONMENT | VIRTUAL_NODE | OP_CONTROL |
                class_type>
object_description ::=    DESCRIPTION informal_text
```

```
implementation_constraints ::=
                IMPLEMENTATION_OR_SYNCHRONIZATION_CONSTRAINTS
informal_text
provided_interface ::= PROVIDED_INTERFACE
                provided_types
                provided_constants
                provided_operations
                provided_set_operations
                provided_exceptions
required_interface ::= REQUIRED_INTERFACE
                required_objects
                required_environment_objects
                required_class_objects
                required_types
                required_operations
                required_exceptions
data flows ::=  data_name direction object object_name
obcs ::=        OBJECT_CONTROL_STRUCTURE
                obcs_description
                obcs_constrained_operations
                <object_code | obcs_implemented_by>
```

O ODS de um objeto permite a reutilização do projeto em diferentes plataformas, pode ser usado para gerar código fonte em Ada ou para gerar Diagramas HOOD, como um processo reverso.

Para a geração de código fonte em Ada, na forma de protótipos, o método HOOD possui uma equivalência entre o ODS e construções da linguagem Ada83, que é uma linguagem baseada em objetos, carecendo de características fortes de linguagens orientadas a objetos, como herança e polimorfismo.

6 Adequação à linguagem C++

No contexto deste trabalho, os aspectos mais interessantes desta notação dizem respeito ao reuso do projeto e à possibilidade de verificação formal. No desenvolvimento do *framework* ao ser adotada C++ como linguagem de implementação através da ferramenta OpenC++, decorreu daí uma série de incompatibilidades entre a notação HOOD e sua equivalência em construções C++.

Um comparativo entre Ada e C++ é apresentado na tabela A1.1, similar ao esquema apresentado por Booch [BOO94], serve para ilustrar as diferenças mais importantes.

Tabela A1.1 - Comparação de características de Ada e C++

Aspecto		Ada	C++
Abstração	Variáveis de instância	sim	sim
	Métodos de instância	sim	sim
	Variáveis de classe	não	sim
	Métodos de classe	não	sim
Encapsulamento	de variáveis	public, private	public, private, protected
	de métodos	public, private	public, private, protected
Modularidade	tipos de módulos	package	file
Hierarquia	herança	não	múltipla
	unidades genéricas	sim	sim
	metaclasses	não	não
Tipagem	tipagem forte	sim	sim
	polimorfismo	não	sim
Concorrência	multitasking	sim	indireta(por classe)

Para contornar estas incompatibilidades, foram tomadas as seguintes decisões:

i - selecionar na notação HOOD/ODS as entidades que encontram uma construção correspondente em C++, desprezando as específicas de Ada. Exemplo: A palavra-chave CONSTRAINED_OPERATIONS refere-se aos pontos de entrada de tarefas em Ada[2], que não possuem equivalência em C++.

ii - adotar as convenções para equivalência entre palavras-chave HOOD e a notação usada no Motf mostradas na Tabela A1.2:

---

[2] *Task entry points*

Tabela A1.2 - Equivalências HOOD/Motf

Notação HOOD	entidade Ada	Notação adotada	entidade C++
CLASS	módulo genérico	Nome.h	arquivo header
OBJECT	cláusula With	CLASS	classe
INTERNALS/OBJECT	cláusula With	OBJECT	instância
OPERATIONS	procedure	OPERATIONS	método

iii - adotar como 'esqueleto' básico para descrição de classes Motf a construção abaixo:

```
CLASS nome_da_classe IS herança_de_classe [;herança_de_classe]
DESCRIPTION comentário
PROVIDED_INTERFACE [-- Descrição dos recursos fornecidos a outros objetos.]
        TYPES [-- Tipos públicos definidos pela classe.]
        CONSTANTS [-- Constantes públicas definidas pela classe.]
        OPERATIONS [-- Métodos públicos definidos pela classe]
        EXCEPTIONS [-- Exceções transmitidas a outros objetos.]
REQUIRED_INTERFACE [-- Descrição dos recursos de outros objetos.]
        CLASS [-- Lista de outras classes Motf usadas]
        ENVIRONMENT_CLASS [-- Lista de outras classes usadas]
        TYPES [-- Lista de nome_da_classe. nome_do_tipo usado]
        OPERATIONS [-- Lista de métodos usados]
        EXCEPTIONS [-- Lista de exceções capturadas]
INTERNALS [-- Descreve os recursos protegidos e privativos da classe]
        OBJECTS [-- Objetos instanciados pela classe]
        TYPES [-- Tipos definidos pela classe.]
        CONSTANTS [-- Constantes definidas pela classe.]
        OPERATIONS [-- Métodos definidos pela classe]
END_CLASS nome_da_classe
```

7 Comentários

A seleção de uma notação única, adequada para a representação de todas as fases de desenvolvimento de software não é trivial. Neste trabalho, a notação foi escolhida tendo como foco principal o domínio - tolerância a falhas e como secundário o modelo - orientação a objetos.

O método HOOD tem sido usado no desenvolvimento de software crítico, principalmente pela sua abrangência (sistemas de grande porte) e disponibilidade de ferramentas automáticas de desenvolvimento que possibilitam a geração de código de programas Ada.

Algumas dificuldades de utilização desse método para representação do *framework* Motf surgiram justamente por ser Ada, uma linguagem representativa do modelo imperativo porém baseada em objetos, o alvo final do método HOOD. Destacam-se a representação de objetos (uma instância) e não de classes (que permitem múltiplas instâncias de idêntica estrutura). Classes em HOOD referem-se a *packages* genéricos, cujas instâncias distinguem-se por suas estruturas de dados e que equivalem a classes genéricas de linguagens de programação orientadas a objetos.

Essas dificuldades foram contornadas com a utilização de conceitos e parte da notação usada por Booch [BOO94], principalmente por ser o método HOOD alicerçado nessa notação.

ANEXO 2

OpenC++

1 Introdução

Neste trabalho, sugere-se que as classes MOTF sejam implementadas na linguagem C++ [STRO91], que não dispõe de metaclasses ou outras facilidades para a implementação de reflexão computacional. Estas facilidades foram buscadas no pré-processador OpenC++ que permite definir meta-objetos como instâncias de classes C++ tradicionais. Em [LIS95a], [LIS95b] podem ser encontrados exemplos implementados com estas ferramentas e executados em ambiente UNIX.

O pré-processador OpenC++ adota um modelo de reflexão computacional com as seguintes características:

a) Meta-objeto: cada objeto, denominado de objeto reflexivo, possui um meta-objeto;

b) Relação: a associação é única entre objetos e meta-objetos, isto é, um meta-objeto não pode ser compartilhado por mais de um objeto;

c) Controle: um meta-objeto pode controlar mensagens dirigidas a seus referentes (chamadas de funções-membro) e acesso à variáveis de instância (variáveis-membro);

d) Todos os meta-objetos descendem da classe MetaObj;

e) Um objeto reflexivo pode ter acesso a seu meta-objeto, através de apontadores.

Um programa em OpenC++ distingue-se de um programa em C++ apenas por suas diretivas, incluidas no texto fonte sob forma de comentários que iniciam por //MOP. O texto resultante é compilado por OCC (compilador de OpenC++), o qual gera um arquivo de trabalho para cada classe reflexiva e dá prosseguimento à compilação, chamando o compilador C++. É possível indicar o compilador a ser usado através da diretiva:

//MOP Compiler = nome-do-compilador

2 Diretivas

As diretivas de OpenC++ [CHI93a], descritas a seguir, são utilizadas para tornar os objetos reflexivos, selecionando os componentes (variáveis e métodos) que devem ser controlados pelo meta nível. As diretivas seguem o formato :

//MOP reflect <item>, onde MOP significa *MetaObject Protocol*.

Tabela A2.1: Descrição das diretivas de OpenC++

Diretiva	Descrição
//MOP reflect class <classe1>,<classe 2>,... : <meta_classe>	Associação de uma ou mais classes da aplicação a uma meta-classe
<tipo de acesso>: //MOP reflect (categoria): <método>; <método>....	Controlar no meta nível chamadas ao(s) método(s) definidos a seguir, com mesmo tipo de acesso (public,protected).
<tipo de acesso> //MOP reflect: <variável>; <variável>....	Controlar no meta nível acessos à(s) variável(is) definidas a seguir, com mesmo tipo de acesso (public,protected).
<tipo de acesso>: //MOP reflect (metamethod): <método>	Substituir o método do nível base por um meta-método, com o mesmo nome do método base

O <tipo de acesso> mencionado na tabela A2.1 pode ser public ou protected, não admitindo ser private.

Exemplo 1: Associação de uma ou mais classes da aplicação a uma meta-classe.

```
//MOP reflect class <classe1>,<classe 2>,... : <meta_classe>
```

```
//MOP reflect class Lagrange, Newton: Motf
```

Define como reflexivas as classes Lagrange e Newton, sendo Motf a classe de seu meta-objeto. Posteriormente devem ser instanciados os objetos reflexivos (nível base) das classes Lagrange e Newton. OpenC++ se encarrega de instanciar o meta-objeto correspondente, por instanciação da (meta) classe Motf.

Exemplo 2: Controlar no meta nível chamadas ao(s) método(s).

```
//MOP reflect (categoria):
    <método>; <método>....
```

```
    public:
//MOP reflect:
    double InterpolaValor (double x, double *abscissa, double
    *ordenada, double& erro);
```

Define como reflexivo o método InterpolaValor (<args>). Em tempo de execução, todas as mensagens dirigidas ao método serão 'desviadas' para o seu meta-objeto.

Exemplo 3: Controlar no meta-nível acessos às variáveis.

```
//MOP  reflect:<variável>; <variável>....
```

```
    public:
//MOP reflect:
    int a;
    Y* y;
```

Especifica como reflexivas as variáveis indicadas. Em tempo de execução, o meta-objeto pode acessar essas variáveis, por exemplo, para manter uma cópia de seus valores, ou mesmo para atualizá-las.

Exemplo 4: Substituir o método do nível base por um meta-método, com o mesmo nome do método de nível base.

```
//MOP reflect (metamethod): <método>
```

```
    public:
//MOP reflect (----------):
    double InterpolaValor(double x, double *abscissa, double
    *ordenada, double& erro);
```

3 Definição de meta-objetos

Meta-objetos podem ser criados como classes comuns de C++, tendo como classe ancestral a classe MetaObj, a qual define os métodos que um meta-objeto pode usar para controlar seu referente. Por ser uma classe comum de C++, um meta-objeto pode receber herança múltipla, i.e., ter uma ou mais classes ancestrais além de MetaObj, e pode sofrer especializações, i.e., pode ser usado como classe base de outras classes descendentes, assim transmitindo indiretamente o protocolo herdado da classe MetaObj.

A computação no meta nível é realizada por redefinição dos métodos adquiridos da classe MetaObj e pelos métodos particulares (definidos ou herdados) da classe do meta-objeto em questão. Os métodos oferecidos pela classe MetaoObj se dividem em três categorias:

i- Métodos que implementam chamadas e acessos à variáveis: são executados quando um método reflexivo é chamado ou uma variável reflexiva é acessada.

Exemplo: O método Meta_MethodCall é executado quando um método reflexivo é chamado. Sua redefinição na classe do meta-objeto é usada para determinar todas as computações a serem realizadas no meta nível por ocasião da interceptação da mensagem.

ii- Métodos usados pelo meta-objeto para executar operações sobre o objeto de nível base: incluem métodos usados para executar um método de nível base, consultar/atribuir valor de variável reflexiva.

Exemplo: O método Meta_HandleMethodCall é usado para executar um método de nível base, cuja chamada foi interceptada.

iii- Métodos diversos usados para operações internas de reflexão, como construtores/destruidores especiais e informações sobre nomes de classes, métodos e identificadores de variáveis de objetos reflexivos.

Exemplo: O método Meta_GetMethodName fornece o nome do método reflexivo cuja identificação é fornecida como argumento.

4 Resumo dos métodos

Os métodos disponíveis na classe MetaObj, a seguir descritos, utilizam os seguintes tipos de argumentos (Tabela A2.2):

Tabela A2.2 - Argumentos da classe MetaObj

Argumento	Significado
Id method_id	Identificador do método reflexivo
Id var_id	Identificador da variável reflexiva
Id category	Nome da categoria
ArgPac& args	Pilha de argumentos de entrada
ArgPac& reply	Pilha de argumentos de saida

Métodos públicos:

- void Meta_MethodCall(Id method_id, Id category, ArgPac& args, ArgPac& reply)

Este método é executado quando um método reflexivo é chamado. Os três primeiros argumentos referem-se ao identificador, categoria e argumentos de chamada do método reflexivo. O argumento reply é um parâmetro valor-resultado e especifica o valor de retorno do método chamado e que será devolvido ao ponto original de chamada (cliente).

- void Meta_Assign(Id var_id, Id category, ArgPac& args)

Este método é executado quando é feito uma tentativa de alteração de uma variável reflexiva. Os dois primeiros argumentos referem-se ao

identificador e categoria da variável e o último contém uma referência ao valor que será atribuído à variável.

- void Meta_Read(Id var_id, Id category, ArgPac& reply)

Este método é executado quando é feito um acesso (consulta) a uma variável reflexiva. Os dois primeiros argumentos referem-se ao identificador e categoria da variável e o último contém uma referência à variável consultada (e não ao seu valor).

- void Meta_StartUp()

Este método é chamado logo após a instanciação de um objeto reflexivo e seu correspondente meta-objeto. É usado para definir construtores especiais a nível de meta-objeto, visto que diversos métodos da classe MetaObj não são disponíveis ao construtor de um meta-objeto.

- void Meta_CleanUp()

Este método é chamado logo após a destruição de um objeto reflexivo e seu correspondente meta-objeto. É usado para definir destruidores especiais a nível de meta-objeto, visto que diversos métodos da classe MetaObj não são disponíveis ao destruidor de um meta-objeto.

Métodos protegidos:

- void Meta_HandleMethodCall(Id method_id, ArgPac& args, ArgPac& reply)

Este método executa o método de nível base identificado pelo primeiro argumento, com argumentos de chamada args e valor de retorno reply. Este método normalmente é chamado pelo método Meta_MethodCall().

- void Meta_HandleAssign(Id var_id, ArgPac& args)

Este método é usado para atribuir o valor obtido em args à variável identificada pelo primeiro argumento. Este método normalmente é chamado pelo método Meta_Assign().

- void Meta_HandleRead(Id var_id, ArgPac& reply)

Este método devolve em reply uma referência ao valor armazenado na variável identificada pelo primeiro argumento. Este método normalmente é chamado pelo método Meta_Read().

- void Meta_AssignValue(Id var_id, ArgPac& args)

Este método é usado para atribuir o valor de args diretamente à variável identificada pelo primeiro argumento. Difere do método Meta_HandleAssign() por atribuir o próprio valor armazenado em args ao invés do valor apontado por args.

- void Meta_ReadValue(Id var_id, ArgPac& reply)

Este método copia o valor da variável identificada pelo primeiro argumento no segundo argumento. Difere do método Meta_HandleRead() por atribuir o próprio valor armazenado na variável reflexiva ao invés de uma referência ao valor.

- const char* Met_GetClassName()

Retorna um apontador para o nome da classe do objeto reflexivo.

- const char* Met_GetMethoName(Id method_id)

Retorna um apontador para o nome do método reflexivo identificado pelo argumento. É oportuno lembrar que um objeto reflexivo pode possuir diversos métodos reflexivos.

- const char* Met_GetVarName(Id method_id)

Retorna um apontador para o nome da variável reflexiva identificada pelo argumento.

5 Argumentos

Uma classe denominada ArgPac é utilizada para armazenamento de argumentos de chamada e valores de retorno de objetos de nível base. Um objeto da classe ArgPac se comporta como uma pilha, capaz de armazenar itens atômicos de dados, a saber: int, char, float e double. Além destes, apontadores e

referências a qualquer tipo, *aliases* de objetos atômicos e *strings* também podem ser armazenados.

A classe ArgPac define os seguintes métodos:

- void PushInt(int)

  Coloca na pilha um argumento do tipo Integer.

- int PopInt()

  Retira da pilha um argumento do tipo Integer.

- void PushDouble(double)

  Coloca na pilha um argumento do tipo Double.

- int PopDouble()

  Retira da pilha um argumento do tipo Double.

- void PushPtr(void*)

  Coloca na pilha um argumento do tipo Pointer.

- void* PopPtr()

  Retira da pilha um argumento do tipo Pointer.

- void PushStr(char*)

  Coloca na pilha um argumento do tipo String. Todo o string é copiado em ArgPac.

- char* PopInt()

  Retira da pilha um argumento do tipo String.

- void PushBin(char*, int)

  Coloca na pilha um argumento do tipo binário. O segundo argumento refere-se ao tamanho, em bytes, do dado binário.

- char* PopBin()

  Retira da pilha um argumento do tipo binário.

Exemplo: Supondo a seguinte associação/ instanciação:

```
//MOP reflect class Interpola: Motf_NVP
refle_Interpola inter;
```

sendo chamada do método reflexivo feita como segue:

```
Y = inter.InterpolaValor(X, eixoX,eixoY,error);
```

No meta-objeto da classe Motf esses argumentos podem ser copiados para variáveis locais:

```
void Moft_NVP:: Meta_MethodCall(id mid, Idcat,ArgPac& args,ArgPac& rep)
{
double X, *abscissa, *ordenada,*erro;
-----------
/* copia os argumentos enviados */
X = args.PopDouble();
abscissa= (double*) argsPopPtr();
ordenada= (double*) argsPopPtr();
erro= (double*) argsPopPtr();
/* recoloca os argumentos na pilha, na ordem original*/
args.PushPtr((void*) erro);
args.PushPtr((void*) ordenada);
args.PushPtr((void*) abscissa);
args.PushDouble(X);
```

Outros tipos de argumentos podem ser utilizados, sendo a aplicação responsável pela conversão e re-conversão, por redefinição dos métodos Pack e Unpack, como a seguir exemplificado.

Exemplo: Seja Estrutura o nome da classe (em C++ definido como Class, Union ou Struct) cuja instância é usada como argumento de chamada de um método reflexivo. Para tornar esse argumento disponível ao meta-objeto, a classe Estrutura deve prover os métodos Pack e Unpack como públicos:

```
Estrutura (ArgPac&)     /* construtor */
/* Inicializa as sua variáveis de instância a partir dos dados do objeto ArgPac */
void Pack (ArgPac&)     /* converte */
/* Copia seus dados no objeto ArgPac */
void Unpack (ArgPac)    /* re-converte */
/* Copia os dados do objeto ArgPac em seus dados de instância */
```

Para facilitar estas operações, OpenC++ fornece alguns métodos [CHI93a].

6 Comentários finais

Os meta-objetos permitem controlar, no meta nível, acesso a variáveis e chamadas de métodos; porém, possui limitações inerentes à extensões de linguagens baseadas em pré-processadores, que não podem utilizar informações geradas (posteriormente) pelo compilador.

O compilador OCC foi obtido por ftp público no endereço:

utsun.s.u-tokyo.ac.jp/lang/openC++

As informações contidas neste anexo foram obtidas das seguintes fontes:

[CHI93a] CHIBA S. **OpenC++ programmer's guide.** Tokyo: Dept. of Information Science, University of Tokyo, 1993. 21 p. (Technical Report n. 93-3).

Este manual faz parte da documentação que acompanha o compilador OCC. Descreve, sem muitos detalhes, os métodos públicos disponíveis na biblioteca de classes OCC. Mostra a utilização de reflexão em programas através de exemplos simples e opções de compilação.

[CHI93b] CHIBA S.; MASUDA T. Designing an extensible distributed language with meta-level architecture. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 1993, Kaiserslautern, Germany, **Proceedings**: Berlin: Springer-Verlag, 1993. p.482-501. (Lecture Notes in Computer Science n. 707).

Este artigo descreve as principais características de OpenC++ e sua utilização na construção de uma aplicação distribuída.

Referências Bibliográficas

[ABB90] ABBOT, R. J. Resourceful systems for fault-tolerance, reliability and safety. **ACM Computing Surveys**, New York, v. 22, n. 3, p. 35-68, Mar. 1990.

[AMB92] AMBLER, A. L. et al. Operational versus Definitional: a perspective of programming paradigms. **IEEE Computer,** Los Angeles, v. 25 n. 9, p. 28-42, Sept. 1992.

[ANC90] ANCONA, M.; DODERO, G. A system architecture for fault tolerance in concurrent software. **IEEE Computer,** Los Angeles, v. 23, n. 10, p.23-32, Oct. 1990.

[ANC95] ANCONA, M.; DODERO, G.; GIANUZZI, V. et al. Reflective architecture for reusable fault-tolerant software. In: LATIN AMERICAN CONFERENCE IN INFORMATICS, 21.,1995, Canela, RS, BR. **Anais**... Porto Alegre: SBC, 1995. p. 87-98.

[AND81] ANDERSON, T.; LEE, P. A. **Fault tolerance, principles and practice**. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[AVI85] AVIZIENIS, A. The n-version approach to fault-tolerant software. **IEEE Transactions on Software Engineering**, New York, v. SE-11 n. 12, p.1491-1501, Dec. 1985 .

[AZE80] AZEREDO, P.A. **Tratamento de exceções em ambientes modulares**. Rio de Janeiro: Depto. de Informática, PUCRJ, 1980. Tese de doutorado.

[BAS84] BASILI, V.; PERRICONE, B. Software errors and complexity: an empirical investigation. **Communications of the ACM**, New York, v. 27 n. 1, p. 42-52, Jan. 1984.

[BEL94] BELENGER, D.; BALANCHANDER, K. Practical Software Reuse: an interim report. In: Frakes, W. (ed.), INTERNATIONAL CONFERENCE IN SOFTWARE REUSE, 3., 1994, Rio de Janeiro, BR. **Proceedings**... Los Alamitos: IEEE, 1994. p. 53-63.

[BIR93] BIRMAN, K. P. The process group approach to reliable distributed computing. **Communications of the ACM**, New York, v. 36, n. 12, p. 36-53, Dec. 1993.

[BIS88] BISHOP, P.G.; PULLEN, F.D. PODS Revisited - A Study of Failure Behavior. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 18., 1988, Tokyo, Japan. **Proceedings**...:New York, IEEE, 1988. p. 02-08.

[BOO94] BOOCH, G. **Object-Oriented Analysis and Design**. Califórnia: The Benjamin/Cummings, 1994. 589p.

[BRI93] BRITO, O. F. G. ; MALUCELLI, V.; LOQUES, O. G. Tolerância a falhas no ambiente RIO. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, 5., 1993, São José dos Campos, SP, BR. **Anais**...: São José dos Campos, INPE, 1993, p. 272-281.

[CAM93] CAMPBELL, R. H. et al. Designing and implementing Choices: an object-oriented system in C++. **Communications of the ACM**, New York, v. 36, n. 9, p. 117-126, Sept. 1993.

[CAR92] CARVALHO, S.E.R. **Exception Handling in Object-Oriented Languages.** Rio de Janeiro: Depto de Informática, PUC-RJ, 1992. 24p. (Monografias em Ciência da Computação, n. 23/92).

[CHI93a] CHIBA S. **OpenC++ programmer's guide.** Tokyo: Dept. of Information Science, University of Tokyo, 1993. 21 p. (Technical Report n. 93-3).

[CHI93b] CHIBA S.; MASUDA T. Designing an extensible distributed language with meta-level architecture. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 1993, Kaiserslautern, Germany, **Proceedings**: Berlin: Springer-Verlag, 1993. p.482-501. (Lecture Notes in Computer Science n. 707).

[CLE93] CLEMATIS, A.; GIANUZZI, V. Structuring conversation in operation/procedure oriented programming languages. **Computer Languages**, Oxford, England, v. 18, n. 3, p.153-168, 1993.

[CLE95] CLEMATIS A.; ANCONA M.; DODERO G. et al. An object-oriented approach to fault tolerant software. In: EUROMICRO WORKSHOP ON PARALLEL AND DISTRIBUTED PROCESSING, 1995, Sanremo, Italy. **Proceedings**... [s.l.]: IEEE, 1995.

[CME88] CMELIK, R.F.; GEHANI, N. H.; ROOME, W.D. Fault-tolerant Concurrent C; a tool for writing fault-tolerant distributed programs. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 18., 1988, Tokyo, Japan. **Proceedings**...New York: IEEE, 1988. p.56-61.

[CRIS82] CRISTIAN, F. Exception Handling and Software Fault Tolerance. **IEEE Transactions on Software Engineering**, New York, v. 31, n. 6, p. 531-540, Jun. 1982.

[CRIS91] CRISTIAN, F. Understanding Fault-Tolerant Distributed Systems. **Communications of the ACM**, New York, v. 34, n.2, p. 56-78, Feb. 1991.

[CUI92] CUI, Q,; GANNON, J. Data-oriented Exception Handling. **IEEE Transactions on Software Engineering**, New York, v. 18, n. 5, p. 393-401, May 1992.

[DoD93] Introducing Ada9x. Office of de Under Secretary of Defense for acquisition, Washington Dc, Feb. 1993.[1]

[FAB95] FABRE, J.C.; NICOMETTE V.; PERENNOU T.; WU, Z. Implementing fault-tolerant applications using reflective OO programming. In: IEEE INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 25., 1995, Pasadena, USA. **Proceedings**...Los Alamitos: IEEE, 1995. p. 489-498.

[FER89] FERBER, J. Computational Reflection in Class Based Object-Oriented Languages. **SIGPLAN Notices**, New York, v. 24, n. 10, p. 317-326, Oct. 1989. Trabalho apresentado no OOPSLA, 1989, New Orleans, Louisiana.

---

[1]This work is sponsored by the Ada9x Project Office under contract F08635-90-C-0066.

[GOL83] GOLDBERG, A. The influence of an object-oriented language on the programming environments. In: ACM COMPUTER SCIENCE CONFERENCE, 1983, Orlando, Florida, USA. **Proceedings**...New York: ACM, 1983. p. 35-54.

[GRA89] GRAUBE, N. Metaclass compatibility. **SIGPLAN Notices** New York, v. 24, n. 10,p. 305-315, Oct. 1989. Trabalho apresentado no OOPSLA, 1989, New Orleans, Louisiana.

[HEC76] HECHT, H. Fault-tolerant software for real-time applications. **A C M Computing Surveys**, New York, v. 8, n. 4, p. 391-408, Dec. 1976.

[HEI92] HEIMERDINGER, W. L. e WEISNSTOCK, C. B. **A Conceptual Framework for System Fault Tolerance**. Pennsylvania: Soft. Eng. Institute, Carnegie-Mellon University, USA, Oct. 1992. 36p. (Technical Report CMU/SEI - 92 - TR-33).

[HOO89] HOOD Reference Manual, Issue 3.0, WME/89, 1989. In: ROBINSON, Peter (ed.), **Object-Oriented Design.** London: Chapman & Hall,1992. Appendix A. p. 151-206.

[JAF91] JAFFE, M. S.; LEVENSON, N. G.; HEIMDAHL, M.; MELHART, B. Software requirements analysis for real-time control systems. **IEEE Transactions on Software Engineering,** New York, v. 17 , n. 3, Mar. 1991 p. 241-258.

[JAG91] JAGANNATHAN, R. e ASHCROFT, E. A. Fault tolerance in parallel implementations of functional languages. IEEE INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 21., 1991, Montréal, Canada. **Proceedings**...Los Alamitos: IEEE, 1991. p. 256-263.

[JAL94] JALOTE, P. **Fault-Tolerance in Distributed Systems.** New Jersey: Prentice-Hall Inc., USA, 1994. 432 p.

[JOH91] JOHNSON, R.E.; FOOTE, B. Designing Reusable Classes. In: PRIETO-DIÁZ, R.; ARANGO, G. (Eds). **Domain Analysis and Software Systems Modeling**. Los Alamitos: IEEE Computer Society Press, 1991. p. 138-149.

[KEL88] KELLY, J.P. et al. A Large Second Generation Experiment in Multi-version Software. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 18., 1988, Tokyo, Japan. **Proceedings**...New York: IEEE, 1988. p. 09-14.

[KEL91] KELLY, J. P. et al. Implementing design diversity to achieve fault-tolerance. **IEEE Software**, Los Alamitos, v. 8 , n. 5, p. 61-71, Jul. 1991.

[KIC91] KICZALES G.; des RIVIERES, J.; BOBROW; D. G. **The Design and Implementation of Meta-Object Protocols**.Cambridge: MIT Press, 1991. 335 p.

[KIM95] KIM, K. H. Challenges in integration of major design techniques for real-time fault-tolerant computer systems. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, 6., 1995, Canela, RS, BR. Palestra convidada.

[KNU87] KNUDSEN, J. L. Better exception handling in block structured systems. **IEEE Software**, Los Alamitos, p. 40-49, May 1987 .

[KOE90] KOENIG, A.; STROUSTRUP, B. Exception Handling for C++. **Journal of Object-Oriented Programming**, New York, v. 3, n. 2, p. 16-33, 1990.

[LAP87] LAPRIE, J.C. et al. Hardware and Software Fault Tolerance: Definition and Analysis of Architectural Solutions. In: FAULT-TOLERANT COMPUTING SYMPOSIUM, 17., 1987, Pittisburg,USA. **Proceedings**...New York: IEEE, 1987. p. 116-121.

[LAP90] LAPRIE, J.C. et al. Definition and Analysis of Hardware- and Software Fault-Tolerant Architectures. **IEEE Computer,** Los Angeles, v. 23, n. 7, p. 39-51, Jul. 1990.

[LAP92] LAPRIE, J.C. (Ed.). **Dependability:** Basic Concepts and Terminology. Vienna: Springer Verlag, 1992.

[LEA93] LEA, R. et al. COOL: system support for distributed object-oriented programming. **Communications of the ACM**, New York, v. 36, n. 9, p. 37-46, Sept. 1993.

[LEE90] LEE, P.A.; ANDERSON, T. **Fault Tolerance**: Principles and Practice. 2nd ed. Berlin: Springer-Verlag, 1990.

[LEV91] LEVESON, N. Software safety in embedded computer systems. **Communications of ACM**, New York, v. 34 , n. 2, p. 34-46.,Feb. 1991

[LIP93] LISBOA, P.H.C.; TEPEDINO, J.F.; MEIRA, S. L. Reflexão computacional em Smalltalk. **Revista Brasileira de Computação**, Rio de Janeiro, v. 7, n. 1, p. 13-24, Jul/Dez 1993.

[LIS94] LISBOA, M.L.; AZEREDO, P. A. Paradigmas de programação: o enfoque de tolerância a falhas. In: XIII JORNADAS DE ATUALIZAÇÃO EM INFORMÁTICA,1994, Caxambu, MG.

[LIS95a] LISBOA, M.L.; CAVALHEIRO, G. C. Reflexão Computacional sobre Técnicas de Tolerância a Falhas em Software. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, 6., 1995, Canela, RS, BR. **Anais**...Porto Alegre: SBC, 1995, p. 405-416.

[LIS95b] LISBOA, M. L.; VENDRUSCOLO, G. Controle de Sensores através de Meta-objetos com Comportamento Inteligente. In: SIMPÓSIO BRASILEIRO DE AUTOMAÇÃO INTELIGENTE, 2., 1995, Curitiba, PR, **Anais**..: Curitiba, CEFET, 1995. p. 219-224.

[MAE87] MAES, P. Concepts and experiments in computational reflection. **SIGPLAN Notices**, New York, v. 22 , n. 12, p. 147-169. Trabalho apresentado no OOPSLA, 1987, Orlando, Flórida.

[MAE88] MAES, P. Issues in computational reflection. In: MAES,P.; NARDI, D. (Ed). **Meta-Level Architecture and Reflection .** Amsterdam: Elsevier Science ,1988. p. 21-35.

[MAL92] MALENFANT, J.; DONY, C.; COINTE, P. Behavioral reflection in a prototype-based language. In: INTERNATIONAL WORKSHOP ON NEW MODELS FOR SOFTWARE ARCHITECTURE/ REFLECTION AND META-LEVEL ARCHITECTURES, 1992,Tokyo, Japan. **Proceedings**...:[s.l.:s.n.], 1992, p. 143-153 .

[MYE76] MYERS, G.J. **Software Reliability** - principles and practices. New York: John Wiley & Sons, 1976. 360 p.

[NAK91] NAKAJO, T. e KUME, H. A case history analysis of software error cause-effect relationships. **IEEE Transactions on Software Engineering**, New York, v. 17, n. 8, p. 830-837, Aug. 1991.

[NAK92] NAKAJIMA, S. What makes a language reflective and how?. In: INTERNATIONAL WORKHOP ON NEW MODELS FOR SOFTWARE ARCHITECTURE'92/REFLECTION AND META-LEVEL ARCHITECTURES, **Proceedings**...Tokyo, Japan, p. 125-136, Nov. 1992.

[MAE88] MAES, P. Issues in computational reflection. In: MAES,P.; NARDI, D. (Ed). **Meta-Level Architecture and Reflection .** Amsterdam: Elsevier Science ,1988. p. 21-35.

[MAL92] MALENFANT, J.; DONY, C.; COINTE, P. Behavioral reflection in a prototype-based language. In: INTERNATIONAL WORKSHOP ON NEW MODELS FOR SOFTWARE ARCHITECTURE/ REFLECTION AND META-LEVEL ARCHITECTURES, 1992,Tokyo, Japan. **Proceedings**...:[s.l.:s.n.], 1992, p. 143-153 .

[MYE76] MYERS, G.J. **Software Reliability -** principles and practices. New York: John Wiley & Sons, 1976. 360 p.

[NAK91] NAKAJO, T. e KUME, H. A case history analysis of software error cause-effect relationships. **IEEE Transactions on Software Engineering**, New York, v. 17, n. 8, p. 830-837, Aug. 1991.

[NAK92] NAKAJIMA, S. What makes a language reflective and how?. In: INTERNATIONAL WORKHOP ON NEW MODELS FOR SOFTWARE ARCHITECTURE'92/REFLECTION AND META-LEVEL ARCHITECTURES, **Proceedings**...Tokyo, Japan, p. 125-136, Nov. 1992.

[PET90] PETRE, M. e WINDER, R. On Languages, Models and Programming Styles. **The Computer Journal**, v. 33, n. 2, p. 173-180, 1990.

[PLA93] PLACE, P. R. H.; KANG, K. C. **Safety-critical software**: status report and annotated bibliography. Pennsilvania: SEI, Carnegie Mellon University, Pittsburgh, Jun. 1993. 78p.

[PUR91] PURTILLO, J. M; JALOTE, P. An environment for developing fault-tolerant software. **IEEE Transactions on Software Engineering**, New York, v. 7, n. 2, p. 153-159, Feb. 1991.

[RAN75] RANDELL, B. System Structure for Software Fault-Tolerance. **IEEE Transactions on Software Engineering,** New York, v. SE-1, n. 2, p. 220-232, Feb. 1975.

[RAN78] RANDELL, B.; LEE, P. A. e TRELEAVEN, P. C. Reliability issues in computing system design. **ACM Computing Surveys**, New York, v. 10, n. 2, p. 123-166, Jun. 1978.

[RUB94a] RUBIRA, C.M.F.; RANDELL, B. Object-oriented environmental software fault-tolerance. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, 6., 1995, Canela, RS, BR. **Anais**...Porto Alegre: SBC, 1995, p. 417-440.

[RUB94b] RUBIRA-CALSAVARA,C.M.F.; STROUD, R.J. Forward and Backward Error Recovery in C++. **Journal of Object-Oriented Systems**, England, n. 1, p. 61-85, Oct. 1994 .

[SCH95] SCHLICHTING, R. D.; THOMAS, V. T. Programming language support for writing fault-tolerant distributed software. **IEEE Transactions on Computers**, New York, v. 44, n. 2, p. 203-212, Feb. 1995.

[SEL91] SELBY, R.; BASILI, V.. R. Analyzing error-prone system structure. **IEEE Transactions on Software Engineering**, New York, v. 17, n. 2, p. 141-152, Feb. 1991.

[SHE92] SHELDON, F. et al. Reliability measurement: from theory to practice. **IEEE Software**, Los Alamitos, v. 9, n. 4, p. 13-20, Jul. 1992.

[SHI91] SHIMEALL, T.; LEVESON, N. An empirical comparison of software fault tolerance and fault elimination. **IEEE Transactions on Software Engineering**, New York, v. 17, n. 2, p. 173-182, Feb. 1991.

[SHR94] SHRIVASTAVA, S. K.; McCue, D.; PARRINGTON, G.D. Structuring Fault-Tolerant Object Systems for Modularity in a Distributed Enviroment. **IEEE Trans. on Parallel and Distributed Systems**, New York, v. 5, n. 4, p.421-432, Apr. 94.

[SHR95] SHRIVASTAVA, S. K. **Lessons learned from building and using the Arjuna distributed programming system.** Berlin: Springer-Verlag,1995. (Lecture Notes on Computer Science n. 938).

[STE88] STEEL, L. Meaning in knowledge representation. In: MAES,P.; NARDI, D. (Ed). **Meta-Level Architecture and Reflection .** Amsterdam: Elsevier Science, 1988. p. 51-59.

[STE94] STEEL, L. Beyond objects. EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 8., 1994, Bologna, Italy. **Proceedings**...Berlin: Springer-Verlag, 1994. p.1-11. (Lecture Notes in Computer Science n. 821).

[STR91] STRIGINI, L.; Di GIANDOMENICO, F. Flexible schemes for application-level fault tolerance. In: SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, 10., 1991, Pisa, Italy, **Proceedings**... New York: IEEE 1991. p. 86-95.

[STRO91] STROUSTRUP, B. **The C++ Programming Language**. 2nd ed., Reading: Addison Wesley, 1991.

[TSI92] TSICHRITZIS, D. **Object Frameworks**. Genéve, Swiss: Centre Universitaire d'Informatique, University of Geneva, p. 31-40, Jul. 1992.

[WAT88] WATANABE,K.; YOZENAWA, A. Reflection in an object-oriented language. **SIGPLAN Notices**, New York, v. 23 , n. 11, p. 306-315.

[WEG92] WEGNER, P. Dimensions od Object-Oriented Modeling. **IEEE Computer,** Los Angeles, v. 25, n. 10, p.12-20, Oct. 1992 .

[WIR73] WIRTH, N. **Systematic Programming:** an introduction. Prentice-Hall, 1973.

[XU94] XU, J. ; RANDELL, B.; RUBIRA-CALSAVARA, C.M.F.et al. Towards an Object-Oriented Approach to Software Fault Tolerance. In: AVRESKY, D.R. (ed.). **Fault-Tolerant Parallel and Distributed Systems**. Los Alamitos: IEEE Computer Society Press, 1994.

[XU95] XU, J. ; RANDELL, B.; ROMANOVSKY, A; RUBIRA C. M. F.; STROUD, R. J. A.; WU, Z. Fault Tolerance in Concurrent Object-Oriented Software Through Coordinated Error Recovery. In: IEEE INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 25., 1995, Pasadena, USA. **Proceedings**...Los Alamitos: IEEE, 1995.

[YOK92] YOKOTE,Y. The Apertos reflective operating system: the concept and its implementation. **SIGPLAN Notices**, New York, v. 27, n. 10,p. 414-434, Oct. 1992. Trabalho apresentado no OOPSLA, 1992, Vancouver, Canada.

Informática UFRGS

MOTF - Meta-Objetos para Tolerância a Falhas

por

Maria Lúcia Blanck Lisbôa

Defesa de Tese apresentada aos Senhores:

Profa. Dra. Cecília Mary Fischer Rubira (UNICAMP)

Profa. Dra. Ingrid Eleonora Schreiber Jansch Pôrto

Prof. Dr. Sérgio Eduardo Rodrigues de Carvalho (PUC/RJ)

Vista e permitida a impressão.
Porto Alegre, 10/ 01/ 96.

Prof. Dr. Paulo Alberto de Azeredo,
Orientador.

Prof José Palazzo Moreira de Oliveira
Coordenador do Curso de Pós-Graduação
em Ciência da Computação - CPGCC
Instituto de Informática - UFRGS