**Implementação de Sistemas Tolerantes a Falhas Usando Programação Reflexiva Orientada a Objetos**

*Sand Luz Corrêa*

Dissertação de Mestrado

# Implementação de Sistemas Tolerantes a Falhas Usando Programação Reflexiva Orientada a Objetos

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Sand Luz Côrrea e aprovada pela Banca Examinadora.

Campinas, 13 de fevereiro de 1998.

Cecília Mary Fischer Rubira
(Orientadora)

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

## FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Corrêa, Sand Luz

C817i Implementação de sistemas tolerantes a falhas usando programação reflexiva orientada a objetos / Sand Luz Corrêa -- Campinas, [S.P. :s.n.], 1997.

Orientador : Cecília Mary Fischer Rubira

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

1. Tolerância a falha (Computação). 2. Programação orientada a objetos (Computação). 3. Framework (Programa de computador). I. Rubira, Cecília Mary Fischer. II. Universidade Estadual de Campinas. Instituto Computação. III. Título.

Instituto de Computação
Universidade Estadual de Campinas

# Implementação de Sistemas Tolerantes a Falhas Usando Programação Reflexiva Orientada a Objetos

Sand Luz Corrêa
Dezembro 1997

**Banca Examinadora:**

Profa. Dra. Cecilia Mary Fischer Rubira (Orientadora)
Instituto de Computação - UNICAMP

Profa. Dra. Maria Lucia Lisboa
Instituto de Informática - UFRGS

Prof. Dr. Luiz Eduardo Buzato
Instituto de Computação - UNICAMP

Prof. Dr. Hans Kurt Edmund Liesenberg (Suplente)
Instituto de Computação -UNICAMP

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas, como requisito parcial para a obtenção do título de mestre em Ciência da Computação

Tese de Mestrado defendida e aprovada em 19 de dezembro de 1997 pela Banca Examinadora composta pelos Professores Doutores

Prof. Dr. Maria Lúcia Lisbôa

Prof. Dr. Luiz Eduardo Buzato

Prof. Dr. Cecília Mary Fischer Rubira

À memória de minha mãe
Divina S. Luz Corrêa

# Agradecimentos

À minha orientadora, Prof ª Cecilia Mary Fischer Rubira, pelo apoio e principalmente confiança para a realização deste trabalho.

À minha família pelo amor e confiança em mim depositados.

A todos meus amigos, em especial, Solange Sobral, Walter Costenaro e Romulo Cesar Fiho. Ao Romulo agradeço também pela parceria em alguns trabalhos, críticas e sugestões mais que valorosas.

À Fundação de Amparo à Pesquisa do Estado de São Paulo - FAPESP - pelo apoio financeiro através da bolsa de mestrado n° 96/0538-3.

# Resumo

Este trabalho tem por objetivo desenvolver uma arquitetura orientada a objetos reflexiva para aplicações tolerantes a falhas de software. Técnicas de orientação a objetos, tais como, abstração de dados, herança e polimorfismo são exploradas, visando obter softwares de melhor confiabilidade e qualidade. Técnicas de reflexão computacional são usadas para estruturar a aplicação, separando de forma nítida os requisitos funcionais da aplicação dos requisitos pertinentes ao domínio de tolerância a falhas. Com isso, nosso objetivo é prover um suporte para tolerância a falhas de software através de técnicas já conhencidas de diversidade de projeto, de forma que este suporte seja incorporado à aplicação da forma menos intrusiva possível, através das técnicas de reflexão computacional. Para maior entendimento e validação dessas técnicas foi desenvolvido um framework orientado a objetos reflexivo e distribuído( FOORD).

# Abstract

The major goal of this work is to develop a reflective object-oriented architecture for software fault-tolerant applications. Object-oriented techniques, such as data abstraction, inheritance and polymorphism are explored to improvement of reliability and quality. Computational reflection techniques are used for structuring applications, so that functional requirements and fault-tolerant requerements can be clearly separated. Thus, our goal is to support software fault tolerance through techniques like design diversity, so that this support can be incorporated to the application in a less intruder way, through computational reflection techniques. For the understanding and validation of these techniques, we developed a reflective object-oriented distributed framework (FOORD).

# Conteúdo

# Lista de Figuras

# Lista de Tabelas

# Capítulo 1
# Introdução

Aplicações de software modernas devem atender a certos requisitos compulsórios para a sua utilização efetiva, como por exemplo, confiabilidade, segurança, distribuição, disponibilidade, adaptabilidade, tolerância a falhas, etc. Exemplos de tais aplicações incluem software para computação móvel, centrais telefônicas, bancos de dados, sistemas distribuídos, controles de trens e de tráfego aéreo e teleconferências. O desenvolvimento de software que atenda a todos esses requisitos simultaneamente é uma tarefa não trivial, e que requer o emprego de técnicas apropriadas durante todo o ciclo de desenvolvimento do software. Em particular, a provisão de tolerância a falhas está baseada na noção de redundância de componentes do sistema, tanto para a detecção de erros quanto para a recuperação dos mesmos. A incorporação de redundância implica em custos maiores de desenvolvimento do sistema. Redundância de hardware é feita através de replicação física de componentes de hardware, enquanto que a redundância de software é alcançada através de redundância de projetos de componentes de software (não simplesmente com sua replicação). Isto implica que a provisão de tolerância a falhas em software é em geral muito mais cara do que em hardware. Redundância de hardware é rotineiramente empregada para aumentar a disponibilidade/confiabilidade de sistemas de computação; entretanto, software é o componente mais crítico de qualquer sistema e portanto, tende a ser tornar o gargalo de confiabilidade de todo sistema. Como conseqüência, o uso de redundância de software implica : (i) num aumento no custo de desenvolvimento de sistemas, e (ii) num aumento da complexidade do sistema por causa da adição dos componentes redundantes. Além disso, a incorporação de tolerância a falhas em software deve ser feita de modo estruturado, cuidadoso e disciplinado; caso contrário, os componentes redundantes podem diminuir, ao invés de aumentar, a confiabilidade do sistema.

Programação orientada a objetos está sendo considerada pela comunidade científica como um modelo de programação efetivo para a produção de software de melhor qualidade. As técnicas orientadas a objetos, se aplicadas de forma apropriada, podem gerar programas com um número menor de erros de software quando comparadas

com as técnicas convencionais de programação estruturada. Benefícios de orientação a objetos incluem abstração de dados, encapsulamento, modularidade, hierarquia, polimorfismo, e reutilização de software. Em particular, o uso de técnicas orientadas a objetos para implementar mecanismos de tolerância a falhas facilita o controle da complexidade do sistema porque elas promovem uma melhor estruturação dos componentes do sistema. Além disso, o uso de orientação a objetos permite a construção de componentes de software reutilizáveis que implementam os mecanismos de tolerância a falhas. Dessa forma, esses componentes podem ser reutilizados numa grande variedade de aplicações correlatas, barateando o desenvolvimento de software tolerante a falhas.

De modo geral, podemos categorizar os requisitos de uma aplicação em dois grupos distintos: **requisitos funcionais** que estão associados com o propósito da aplicação; e os **requisitos administrativos** relacionados em fornecer infra-estrutura para realização deste propósito. Tolerância a falhas é um requisito administrativo e como tal, deveria ser incorporada à aplicação de forma o mais transparente possível para o programador, de modo a facilitar a construção de aplicações tolerantes a falhas. De preferência, sem alterar a estrutura original da aplicação, isto é, a incorporação de redundância deveria ser o menos intrusiva possível.

A idéia de se usar reflexão computacional para estender a semântica da máquina virtual do sistema de forma transparente não é nova. Na literatura existem vários experimento práticos, como veremos no Capítulo 6, que comprovam a viabilidade do uso de reflexão computacional. Reflexão computacional foi incorporada ao modelo de objetos na forma de protocolo de metaobjetos, que é uma interface entre a máquina virtual do sistema e o programador da aplicação que permite estender ou modificar a semântica da própria máquina virtual para que esta se ajuste às necessidades específicas da aplicação. A programação orientada a objetos reflexiva encoraja uma organização modular de sistemas através da separação nítida entre objetos da aplicação e objetos(metaobjetos) que gerenciam e controlam a aplicação. No contexto desta dissertação, reflexão é usada para implementar diferentes estratégias de tolerância a falhas no meta-nível de modo transparente e não intrusivo. O objetivo é implementar um framework reflexivo orientado a objetos para o desenvolvimento de software tolerante a falhas. Esperamos que os sistemas desenvolvidos usando este framework apresente as seguintes vantagens:

(i) separação das ações pertinentes ao domínio da aplicação da ações específicas de tolerância a falhas;
(ii) reutilização de objetos tolerantes a falhas;
(iii) adequação das estratégias de tolerância a falhas à aplicação;
(iv) maior adaptabilidade dos mecanismos de tolerância a falhas utilizados pelo sistema.

O termo framework empregado nesta dissertação denota uma arquitetura de software semi-acabada que consiste de vários componentes individuais e interconexões entre eles, de forma a criar uma infra-estrutura de suporte pré-fabricada para o desenvolvimento de aplicações de um domínio específico[23]. O framework foi

desenvolvido em OpenC++[11], um pre-processador para C++ com suporte para reflexão computacional e o ambiente de programação distribuída Arjuna, com suporte para ações atômicas. Para demonstrar a aplicabilidade e viabilidade da abordagem proposta para tratar tolerância a falhas, implementamos um pequeno estudo de caso no Capítulo 5.

O restante desta dissertação está organizada como se segue: o Capítulo 2 provê informações básicas e terminologias relacionadas com o modelo de objetos. O Capítulo 3 descreve alguns conceitos fundamentais de tolerância a falhas. O Capítulo 4 apresenta o conceito de reflexão computacional, arquitetura reflexiva, principais usos e o estudo de algumas linguagens reflexivas. O Capítulo 5 apresenta o projeto e implementação do framework FOORD e as classes que o compõe. Finalmente, o Capítulo 6 apresenta algumas conclusões, trabalhos relacionados e sugestões futuras.

# Capítulo 2

# Fundamentos de Orientação a Objetos

A abstração mais comum em computação é feita através de linguagens de programação. Estas, por sua vez, exercem uma influência inegável na maneira como formulamos nossos pensamentos. Como conseqüência, a linguagem tem forte influência na confiabilidade e legibilidade de sistemas, sendo portanto, um dos pontos iniciais para se construir softwares mais confiáveis. Neste contexto, o papel da abstração em linguagens de programação foi crucial para o desenvolvimento de softwares.

O primeiro grande passo para a construção de softwares mais confiáveis foi a introdução de abstrações de dados e controle em linguagens de programação. Como exemplo podemos citar: funções e subrotinas em Fortran, descrição de dados em Cobol e estruturas de blocos e procedimentos em Algol. O próximo passo foi o aparecimento do conceito de objeto, no final da década de 60, introduzido por Simula67[17] e consolidado posteriormente por Smalltalk-80[22]. O conceito de objeto aprimorou o conceito de macro-instrução como componente reutilizável, encapsulando dados e ações na forma de objetos. Este mecanismo de encapsulamento aumentou, por sua vez, o poder de abstração das linguagens de programação e o paradigma de orientação a objetos tornou-se um sucesso para estruturar e construir sistemas complexos.

O paradigma de orientação a objetos constitui-se numa coleção de conceitos cujo objetivo principal é prover componentes de software reutilizáveis. Isto é obtido através da noção de objetos, classes, herança, ligação dinâmica e polimorfismo. A Seção seguinte introduz estes conceitos e estabelece a terminologia que será empregada no restante desta dissertação. A Seção 2.2 apresenta os conceitos de framework e padrões de projeto. A Seção 2.3 discute metodologia orientada a objetos, em particular a UML, adotada nesta dissertação; para finalizar, a Seção 2.4 traz um resumo sobre este capítulo.

## 2.1 Conceitos Básicos

Esta Seção tem por objetivo definir alguns termos comumente usados no modelo de objetos bem como sua representação. A metodologia utilizada para a notação destes conceitos é a UML.

### Objetos

Objetos são entidades que encapsulam dados e operações. Dados, ou atributos, representam o estado do objeto. Operações, ou métodos, representam o seu comportamento. A interface de um objeto é o conjunto de operações que podem ser requisitadas por outros objetos. A comunicação entre objetos é feita através de mensagens que identificam operações em objetos. Tais operações devem estar disponíveis na sua interface. Objetos são representados por retângulos similares às classes, porém o nome do objeto no retângulo é sublinhado, uma forma de diferenciá-lo de classes. O nome da classe do objeto pode ser mencionada, após o nome do objeto e separados pelo símbolo ":", como mostra a Figura abaixo:

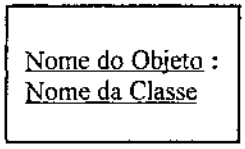



Figura 2.1: Notação para Objetos

### Classes

Classe é uma abstração que permite classificar objetos segundo suas propriedades e comportamento. Objetos com propriedades e comportamentos similares são agrupados numa mesma classe. Assim, uma classe contem a definição dos atributos e métodos dos objetos. Portanto, o termo objeto designa uma instância de uma classe. Uma classe é representada por um retângulo de linhas sólidas, podendo haver três compartimentos, contendo respectivamente: o nome da classe, uma lista de atributo e uma lista de operações, como mostra a figura abaixo:

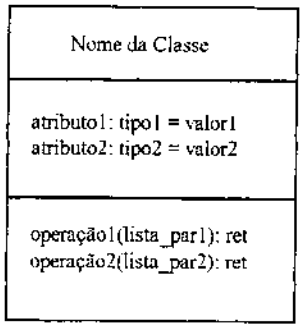



Figura 2.2: Notação para Classes

**Relacionamentos**

Uma ligação é uma conexão estrutural entre objetos de diferentes classe. Uma relacionamento é um conjunto de ligações com estruturas e semânticas comuns, assim como uma classe é um conjunto de objetos similares. Na UML existem seis tipos de relacionamentos, como mostra a figura abaixo:

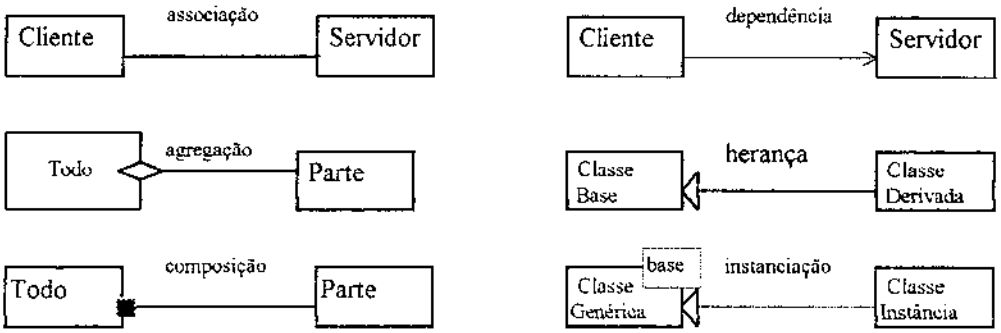



Figura 2.3: Tipos de Relacionamentos

Uma **associação** indica que a classe cliente usa algumas facilidades de uma classe servidor. Os tempos de vida dos objetos que participam do relacionamento são independentes. Uma **agregação** representa um relacionamento "todo/parte" onde objetos representando componentes são associados com um objeto representando o todo. A agregação é representada por um diamante vazio no objeto que representa o todo no relacionamento. Uma **composição** é uma forma de agregação onde os relacionamentos entre as partes são válidos apenas dentro da composição. Logo, o tempo de vida dos componentes depende do tempo de vida do todo. A composição é representada como a agregação, porém com um diamante cheio. Uma **generalização** é o relacionamento entre uma classe e uma ou mais de suas versões refinadas. A classe sendo refinada é chamada

superclasse ou classe base e cada versão é chamada subclasse ou classe derivada. Cada subclasse herda características de sua superclasse. A generalização é representada por uma linha direcionada , da subclasse para a superclasse. Uma **dependência** significa que uma classe depende de algum serviço de uma outra classe, mas não tem uma referência interna ou ponteiro para a mesma. Um relacionamento **instância** ocorre entre uma classe genérica e a classe que resulta da criação da instância.

Outro tipo de relacionamento que usamos nesta dissertação mais que não possui representação específica na UML é o relacionamento entre classe e metaclasse. Para representá-lo adotamos a notação ilustrada na Figura 2.4:


Figura 2.4: Notação para representar o relacionamento entre uma classe e sua metaclasse

**Tipo e Tipos Abstratos de Dados**

Um tipo é a descrição de uma interface que especifica o comportamento comum a todos objetos de um determinado tipo. Por exemplo, uma classe especifica uma implementação particular de um tipo. Um tipo pode ser associado a um atributo: estática ou dinamicamente. Tipagem estática ocorre quando o tipo de uma variável referindo-se a um objeto é verificado antes do tempo de execução. C++ e Eiffel são linguagens com tipagem estática. Tipagem dinâmica ocorre quando o tipo da variável referindo-se a um objeto é conhecido em tempo de compilação e pode ser alterado em tempo de execução. Isto ocorre em Smalltalk e Objective-C.

O conceito de tipos abstratos de dados evoluiu do conceito de abstração de dados o qual provê abstrações de estruturas através de interfaces bem definidas. Abstração de dados é uma técnica para esconder informações como forma de tratar complexidade. Tipos abstratos de dados estendem o conceito de abstração de dados separando-se sua especificação de sua implementação. Esta separação reflete diretamente na sintaxe e semântica da linguagem. Assim, para usar um tipo abstrato de dados basta saber sua especificação; não há necessidade de se conhecer sua implementação. Tipos abstratos de dados consistem de duas partes como mostra a Figura 2.5:

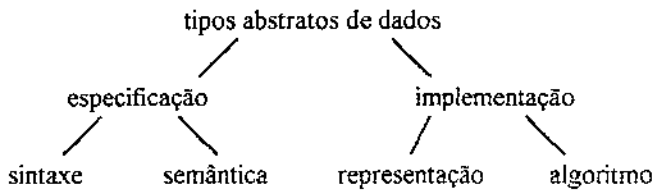



Figura 2.5: Estrutura de um Tipo Abstrato de Dados

A especificação de um tipo abstrato de dados descreve sua sintaxe e semântica associada. A sintaxe geralmente refere-se à assinatura do tipo abstrato de dados e define sua interface. Algumas linguagens permitem especificar também a semântica de um tipo de dados. A implementação de um tipo abstrato de dados descreve sua representação em termos de estruturas de dados primitivas (representação) e os algoritmos associados a cada operação (algoritmos).

**Encapsulamento**

Segundo Snyder[52], encapsulamento é uma técnica para minimizar interdependências entre módulos através de interfaces externas bem definidas. Um módulo é encapsulado se seus clientes conseguem acessá-lo apenas através de sua interface externa. Portanto, encapsulamento provê uma forma mais segura de efetuar modificações em código, facilitando sua manutenção. Em uma linguagem orientada a objetos tradicional, a definição de uma classe é um módulo cuja interface externa consiste de um conjunto de operações; mudanças na implementação da classe, mas que preservem a sua interface externa, não afetam o código fora da definição da mesma.

Entretanto, o mecanismo de herança introduz um novo tipo de cliente para uma classe: sua subclasse. Deve-se então verificar também a interface externa que será fornecida para a classe derivada. As linguagens orientadas a objetos têm diferido na forma como elas permitem o acesso às interfaces externas por classes ordinárias e subclasse. Algumas linguagens introduzem os seguintes conceitos de visibilidade:

- Pública: todo método ou atributo declarados como públicos podem ser acessados diretamente por qualquer cliente.
- Privada: todo método ou atributo declarados como privados não podem ser acessados, manipulados ou invocados diretamente por nenhum cliente.
- Visibilidade de subclasse: se um atributo ou método é declarado com esta visibilidade, eles só podem ser acessados ou invocados diretamente por subclasses.

C++ e Java são exemplos de linguagens que possuem os três níveis de visibilidade definidos acima. Entretanto, em Java existe, além da própria classe, outra unidade de encapsulamento denominada ``package"(grupo de classes em um mesmo arquivo ). Todo método ou atributo não privado de uma classe pode ser acessado ou invocado diretamente por outras classes definidas no mesmo ``package". Em Smalltalk e em CLOS atributos só podem ser acessados através de métodos. Entretanto, operações nestas linguagens são visíveis para qualquer cliente, ou seja, existe apenas a visibilidade pública. Uma subclasse portanto, herda todos os atributos e métodos de sua superclasse. Entretanto, em CLOS, classes não encapsulam funcionalidade, ou seja, métodos são funções ordinárias que se aplicam sobre objetos fornecidos como argumentos. As Tabelas

2.1 e 2.2 exemplificam a visibilidade, fornecida pelas linguagens discutidas acima, para classes ordinárias e subclasses respectivamente.

| | Visibilidade de atributos | Visibilidade de métodos |
|---|---|---|
| C++ | dados públicos | apenas operações públicas |
| CLOS | dados acessados por métodos | todos métodos visíveis |
| Smalltalk | dados acessados por métodos | todos métodos visíveis |
| Java | dados públicos e dados não privados definidos no mesmo "package" | Idem atributos |

Tabela 2.1 Visibilidade de atributos e métodos para classes ordinárias

| | Visibilidade de atributos | Visibilidade de métodos |
|---|---|---|
| C++ | dados públicos e protegidos | métodos públicos e protegidos |
| CLOS | dados acessados por métodos | todos métodos visíveis |
| Smalltalk | acesso direto aos dados | todos métodos visíveis |
| Java | dados públicos, não privados definidos no mesmo "package" e dados protegidos | Idem atributo |

Tabela 2.2: Visibilidade de atributos e métodos para subclasses

**Herança**

Herança é o mecanismo que permite definir novas classes a partir de classes já existentes. Através deste mecanismo, classes podem herdar atributos e métodos de outras classes, bem como modificar ou criar novos métodos. Herança é um mecanismo de especialização de classes e como conseqüência, uma hierarquia é formada. Classes que herdam de outras classes são designadas subclasses e classes herdadas são chamadas superclasses. De acordo com Liskov[30], podemos obter dois tipos de hierarquias através do uso de herança, a saber, hierarquia de implementação e hierarquia de tipo.

Na hierarquias de implementação, herança é usada como uma técnica para implementar tipos abstratos de dados similares a outros já existentes. O objetivo principal é reutilizar código já existente e não há nenhuma garantia que a subclasse tenha o mesmo comportamento de sua superclasse. Isto pode levar a alguns problemas, como a subclasse herdar comportamentos(operações) indesejáveis. Portanto, este tipo de herança não é recomendável, uma vez que ela pode dar origem a comportamentos incorretos. A hierarquias de tipo é composta por subtipos e supertipos. Um tipo **S** é um subtipo de **T** se e somente se **S** provê pelo menos o comportamento de **T**. Um objeto de **S** pode então ser usado como se fosse do tipo **T** pois garante-se que ele provê pelo menos as operações contidas em **T**. Este tipo de herança relaciona o comportamento de dois tipos e não necessariamente suas implementações, e portanto, existe uma relação verdadeira de generalização/especialização.

Até então tratamos apenas de herança simples, ou seja, cada subclasse possui apenas uma única superclasse imediata. Logo, a hierarquia de classes com herança simples é uma árvore. Entretanto, existem situações em que é desejável que uma classe possua mais de uma superclasse imediata. Neste caso, temos herança múltipla. Como uma classe pode ter mais de um predecessor imediato, a hierarquia de classes com herança múltipla é uma gráfico orientado acíclico. Uma conseqüência imediata do uso de herança múltipla é que superclasses podem possuir atributos ou métodos com nomes iguais. Surge então na subclasse o problema de saber exatamente a que superclasse refere-se o atributo ou método associado ao nome em questão. Algumas linguagens, para resolver o problema, ordenam uma lista de superclasses, na qual o atributo ou método conflitante pertence sempre à classe mais específica. Outra estratégia usada, por exemplo em C++, trata-se de acrescentar o nome da classe ao nome do atributo ou método conflitante. Esta abordagem é mais flexível que a anterior, uma vez que o programador pode escolher qual atributo ou método a usar.

**Ligação Dinâmica**

Em geral, ligação[1] é uma associação entre um atributo e uma entidade (variáveis, métodos área de armazenamento). Uma ligação estática ocorre antes do tempo de execução e permanece inalterada durante toda a execução do programa. Por outro lado, uma ligação dinâmica ocorre em tempo de execução e portanto pode ser alterada durante a execução do programa, provendo alto grau de flexibilidade de programação. Em programação orientada a objeto, ligação dinâmica refere-se ao mapeamento entre o nome do método e sua implementação, isto é, uma mensagem invocada em tempo de execução é associada dinamicamente a uma implementação que depende da classe do objeto que invocou a mensagem. Por exemplo, em C++ o programador deve requisitar uma ligação dinâmica para uma mensagem explicitamente declarando-a como um método virtual na classe base e redefinindo-a na classe derivada. Quando este método é invocado, ele será associado à implementação definida na classe base se o tipo do objeto que a invocou for da classe base; ele será associado à implementação definida na classe derivada se o tipo do objeto que a invocou for da classe derivada.

**Polimorfismo**

Muitas linguagens de programação são monomórficas, uma vez que valores estão associados a um único tipo. Esta restrição resultou no estudo de técnicas polimórficas como forma de prover disciplinas de tipos mais flexíveis às linguagens de programação. Portanto, podemos definir polimorfismo como a habilidade de um valor ter mais de um tipo.

Segundo a taxinomia de técnicas polimórficas de Cardelli e Wegner[10], mostrada na Figura 2.6, polimorfismo pode ser classificado em: universal e ad hoc. Ad

[1] Do inglês *binding*

hoc é aquele em que as técnicas polimórficas aplicam-se somente sobre um número específico de tipos, de forma não sistemática. Ao contrário, em polimorfismo universal, as técnicas polimórficas aplicam-se a um conjunto infinito de tipos, de forma sistemática.

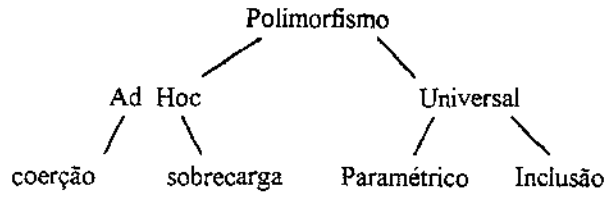



Figura 2.6: Classificação de polimorfismo segundo Cardeli e Wegner

Polimorfismo ad hoc classifica-se em:

- coerção: é uma técnica de polimorfismo que permite a conversão ou mapeamento interno entre tipos diferentes. Por exemplo, se uma operação é definida sobre dois reais e um inteiro e um real são fornecidos como parâmetros, então o inteiro será mapeado para real.
- sobrecarga: é uma técnica de polimorfismo que permite que o nome de uma operação seja usada mais de uma vez com tipos diferentes de parâmetros.

Polimorfismo universal classifica-se em:

- paramétrico: uma operação única (codificada uma única vez) será aplicada uniformemente sobre um intervalo de tipos. Funções paramétricas são também chamadas funções genéricas. Um exemplo de polimorfismo paramétrico são ``templates'' em C++.
- inclusão: também permite uma operação atuar sobre um intervalo de tipos. Entretanto, este intervalo é determinado pelos relacionamentos de tipo-subtipo. Com o polimorfismo de inclusão, uma operação definida sobre um tipo particular, irá também operar sobre quaisquer subtipos. C++ implementa polimorfismo de inclusão.

Aparentemente, parece que existe um conflito entre a flexibilidade oferecida por linguagens polimórficas e a correção provida por checagem de tipos. Entretanto, é possível fornecer tal flexibilidade e ainda garantir correção de tipos. É possível definir-se um método para um conjunto de tipos diferentes, com cada tipo determinando uma interpretação para o método. Isto é possível pois, por um lado, a checagem de tipo garante que existe uma interpretação para um tipo particular, por outro, a existência de ligação dinâmica resolve a interpretação exata para aquele tipo.

**Delegação**

Delegação é um mecanismo parecido com herança mas que opera diretamente entre objetos e não entre classes. Quando usamos delegação os objetos são vistos como protótipos que delegam seus comportamentos para outros objetos. Um objeto então, consiste de duas partes: uma parte particular, que contém o que é apenas do objeto, e uma parte compartilhada, que contem a lista de protótipos os quais o objeto compartilha alguns comportamentos. Toda mensagem delegada carrega o nome de seu delegador. Delegação é o mecanismo de compartilhamento usado em linguagens que não possuem o conceito de classe (Self[56]e Actor[1]).

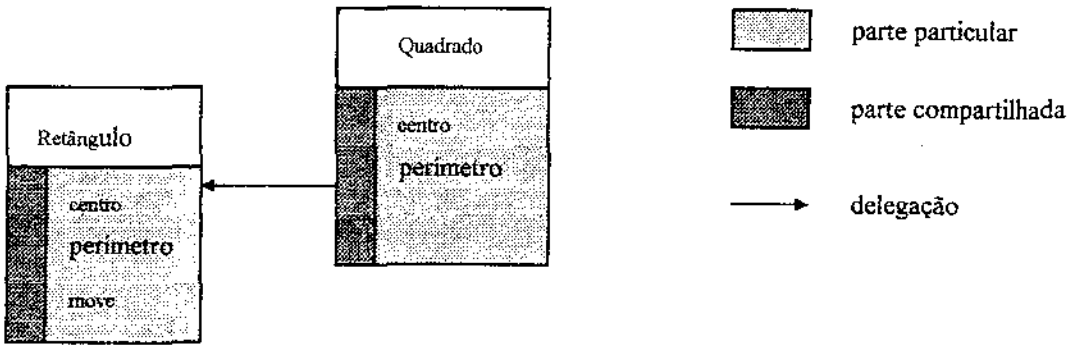



Figura 2.7: Estrutura de um objeto e seu proxy

Considere, por exemplo, a Figura 2.7 que ilustra dois objetos: `Retângulo` e `Quadrado`. O objeto `Quadrado` possui seus próprios valores para os atributos `centro` e `perímetro` e compartilha com o objeto `Retângulo` a operação `move()` que move o centro da figura. Portanto, `Retangulo` é denominado o ``proxy" de `Quadrado`. Então, quando `Quadrado` recebe uma mensagem move, este método é procurado localmente. Como em `Quadrado` não existe nenhuma operação com este nome, a mensagem é delegada a seu ``proxy".O objeto `Retangulo` recebe a mensagem delegada e esta é executada. Quando o método `move()` necessitar do valor de centro ele delega a mensagem centro de volta para seu cliente(`Quadrado`). O objeto `Retangulo` sabe quem é seu cliente, uma vez que este é fornecido como parâmetro da mensagem delegada. Assim, todo atributo ou método invocado em `Retangulo` durante a execução de `move()` é retornada para o objeto `Quadrado`. Se estes atributos ou métodos não estiverem definidos em `Quadrado`, a mensagem é delegada novamente a `Retangulo`, que usa então, a sua própria definição.

De maneira geral, apesar de delegação ser um mecanismo mais poderoso que herança, a escolha de qual estratégia adotar depende da aplicação. Algumas aplicações necessitam da flexibilidade e dinamismo oferecido pelo mecanismo de delegação com compartilhamento a nível de objetos e controlado explicitamente pelo programador.

Outras aplicações entretanto, necessitam da segurança fornecida pela forma estática com que classes são relacionadas a suas superclasses em herança, onde os mecanismos de compartilhamento são implícitos e aplicam-se uniformemente a todos os membros do grupo.

**Metaclasses**

O paradigma orientado a objetos dá apoio a três tipos de abstrações: (i) abstração de dados para comunicação de objetos; (ii) super-abstração para compartilhamento de comportamento(hierarquias); e (iii) meta-abstração para a descrição da própria abstração. As duas primeiras abstrações foram discutidas anteriormente. Vamos então discutir sobre meta-abstração. Mais especificamente, metaclasses são classes cujas instâncias são também classes [7]. Como vimos anteriormente, uma classe é a descrição do que é comum em um grupo de elementos denominados instâncias. Todas instâncias de uma classe compartilham a mesma estrutura e comportamento. Apenas estados individuais são próprios de cada instância. A classe fornece, então, um repositório para o código compartilhado e a estrutura para criar novas instâncias. Por outro lado, uma classe é ela própria uma instância e como tal, deve haver uma classe que a descreva. A classe que contem a descrição de uma outra classe é chamada metaclasse, e como tal, ela provê o protocolo para criação e inicialização de suas instâncias: outra classe. Como conseqüência, uma metaclasse pode ser usada para armazenar informações globais de todos os objetos de uma classe, bem como controlar a criação de novas instâncias da classe. A relação entre metaclasse, classe e instância é ilustrada na Figura 2.8.

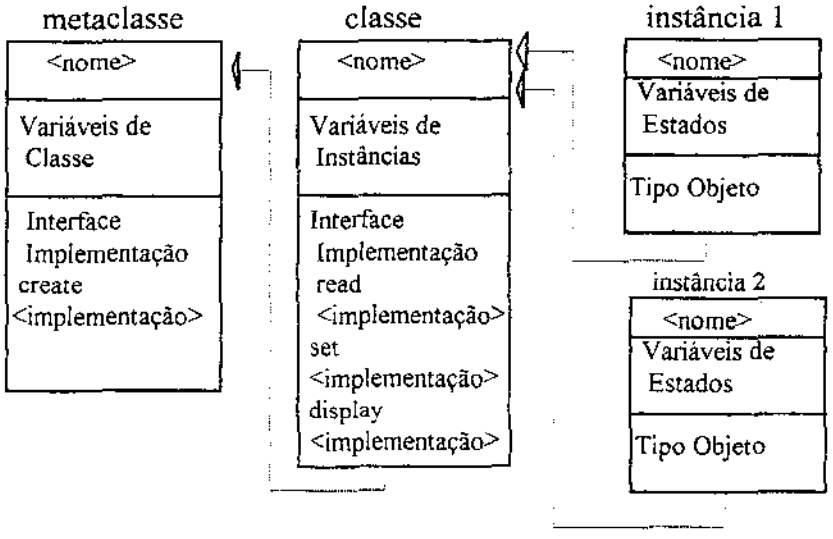



Figura 2.8: Relação entre instâncias, classe e metaclasse

Em alguns sistemas, estas estruturas estão misturadas e muitas vezes a classe se confunde com a metaclasse. Entretanto a idéia de separá-las em estruturas diferentes é particularmente interessante uma vez que uma classe contém metadados que são aplicados a cada uma de suas instâncias. Neste caso, metaclasses permitem associar

metaobjetos aos objetos de uma classe. Essa associação é aproveitada em reflexão computacional da seguinte maneira: (i) um metaobjeto ^x representa os aspectos estruturais e comportamentais de um objeto x; (ii) existe uma conexão causal[2] entre x e ^x; (iii) qualquer alteração em ^x é automaticamente refletida em x.

**Reflexão Computacional**

Reflexão computacional é um mecanismo que permite a um sistema manipular ou modificar seu comportamento devido a incorporação de estruturas representando seu próprio estado[32]. Este conceito, aplicado no contexto de linguagens orientadas a objeto, permite que atributos como invocação de mensagens, interfaces e herança possam também ser objetos de computação do próprio sistema. A combinação de técnicas orientadas a objeto e reflexão tem sido feita através de protocolos de metaobjetos (PMO). Cada objeto é associado a um metaobjeto que guarda informações sobre a estrutura e o comportamento do objeto e fornece meios pelos quais estas informações podem ser modificadas. Todo acesso ao objeto é interceptado pelo metaobjeto de forma que este possa executar várias tarefas para o objeto a ele associado. Como resultado, é possível modificar os mecanismos pelos quais objetos e classes são definidos. Reflexão computacional é portanto uma técnica promissora para o desenvolvimento de sistemas adaptativos e será investigada com mais detalhes no capítulo 4.

## 2.2 Frameworks e Padrões de Projeto

Embora classes abstratas provêem uma forma de expressar o projeto de uma classe, o conceito de classe não oferece a granulosidade necessária para alcançar reutilização de software em grande escala. Para alcançar tal propósito, surgiu o conceito de framework. Um framework é um conjunto de classes abstratas e concretas, bem como suas interconexões, que provê uma infra-estrutura genérica de soluções para um conjunto de problemas[23,64]. A base para sua construção são classes abstratas, através das quais outros componentes, isto é, classes concretas ou classes abstratas podem ser implementadas baseando-se em contratos oferecidos pelas classes abstratas. Em geral, um framework comprime um software genérico para um domínio de aplicação específico. Aplicações baseadas em frameworks são construídas customizando-se algumas classes de forma que o framework antecipe uma parte do projeto do software. Mas, ao contrário de bibliotecas de classes convencionais, como mostra a Figura 2.9, o controle da aplicação (ou fluxo de evento) é feito em tempo de execução pelo próprio framework e não pela aplicação. Programadores que utilizam um framework implementam um código que será chamado pelo framework, ao invés de implementarem código que invoca bibliotecas de classe. Como conseqüência, com frameworks, as aplicações reutizam o fluxo de controle e a arquitetura que ele provê como um todo, enquanto que componentes de bibliotecas de classes são usados individualmente.



[2] Do inglês *casual conection*

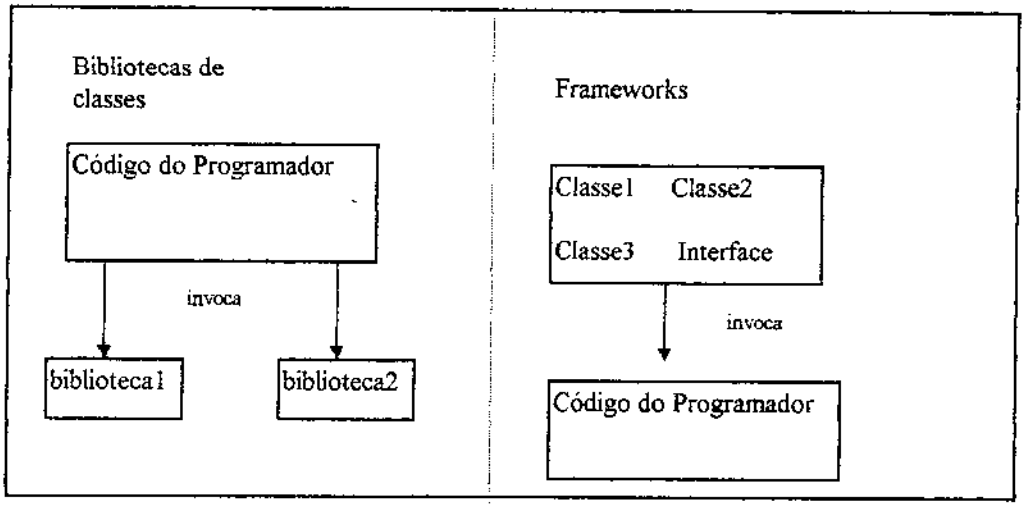



Figura 2.9: Diferença entre Bibliotecas e Frameworks

Dentre as vantagens providas por um framework podemos citar:

- infra-estrutura pré-fabricada. Frameworks reduzem a codificação, depuração e teste de aplicações pela provisão de um subsistema operante.
- arquitetura de apoio. Frameworks fornecem uma arquitetura pronta para ser usada. O programador precisa apenas saber como usá-la para obter o comportamento desejado. Ele não precisa se preocupar com a construção de algumas classes, conexões entre elas, quais métodos estão disponíveis e em que ordem devem ser invocados. O framework oculta estas complexidades fornecendo um alto nível de abstração.
- aplicações menos monolíticas. O uso de frameworks encoraja a implementação de porções de códigos pequenos, através da utilização de várias funções fornecidas por frameworks diferentes. Ao contrário de se escrever uma única aplicação monolítica, escreve-se porções pequenas de código que executam em frameworks diferentes.
- redução do custo de manutenção

Em suma, os benefícios provenientes do uso de frameworks estão relacionados com o alto nível de reutilização de código para o desenvolvimento de sistemas complexos. A maior desvantagem do seu uso é que eles limitam a flexibilidade, uma vez que os componentes da aplicação devem seguir as restrições impostas pelo framework. Mas em geral, o projeto de frameworks bem estruturados têm contribuído para a solução de várias aplicações complexas.

**Padrões de Projeto**

Em geral, padrões auxiliam na redução de complexidade em várias situações reais. Por exemplo, em algumas ocasiões, a seqüência de ações para executar uma tarefa pode ser crucial. Ao invés de se escolher uma ação de um conjunto de infinito de combinações possíveis os padrões oferecem uma solução para o problema já testada e que funciona. Em programação e projeto de software não é diferente. Constantemente, programadores estão reutilizando código escrito por outros. Esta reutilização envolve a observação do padrão de um outro código e sua adaptação para o programa. Padrões de projeto são abstrações dessa atividade de reutilização, ou seja, eles constituem um conjunto de regras para descrever e acomodar certas tarefas dentro de um software em desenvolvimento.

Em [21] é apresentado um catálogo de padrões de projeto orientados a objetos, independentes de linguagens de programação ou aplicações. O catálogo descreve os padrões através de quatro elementos essenciais: **o nome do padrão**; **o problema**, que descreve quando o padrão deve ser aplicado; **a solução**, que descreve o elementos(classes e objetos) que constituem o projeto, seus relacionamentos e responsabilidades; e **as conseqüências**, que são os resultados e as decisões que decorrem da utilização do padrão.

A Figura 2.10 mostra um exemplo de um padrão de projeto encontrado em [21]. O padrão Estado (Figura 2.10) é aplicado quando é preciso definir uma dependência entre objetos de forma que, quando o estado de um objeto muda, todos seus dependentes são notificados e atualizados automaticamente. Os objetos destes padrão são `Observado` e `Observador`. Um `Observado` pode estar associado a vários objetos `Observador` e todos `Observador` são notificados quando o `Observado` muda de estado. Assim, cada `Observador` consulta o `Observado` para sincronizar seu estado.

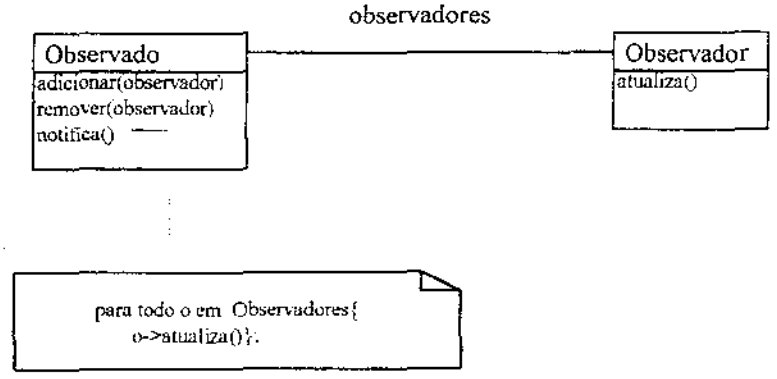



Figura 2.10: Exemplo de Padrão de Projeto (Padrão Estado)

Os conceitos de padrões e frameworks são freqüentemente associados. Padrões podem ser vistos como descrições abstratas de frameworks, que facilitam a reutilização de arquiteturas de software e não dependem de linguagens de programação. Por outro lado, frameworks podem ser vistos como a realização concreta de padrões e são implementados em uma linguagem particular. Nesta dissertação, padrões são usados para documentar a forma e o conteúdo do framework proposto.

**Metapadrões**

Metapadrões[40] consistem na especificação de um conjunto de padrões que descrevem como construir frameworks independentes de um domínio especifico. Metapadrões são usados para classificar e descrever padrões de projeto e, portanto, não substituem as abordagens de padrões, mas as complementam.

A abordagem de Metapadrões identifica dois tipos de métodos para a implementação de frameworks: método template e método hook. Estes métodos formam os metapadrões necessários para projetar frameworks. Os métodos templates implementam a parte fixa do framework enquanto que os métodos hooks implementam a parte adaptável. Em outras palavras, os métodos hook correspondem ao métodos abstratos (que devem ser "preenchidos") e aos métodos que podem ser substituídos. Os métodos templates correspondem aos métodos que invocam os métodos hook. A Figura 2.11 é um exemplo do uso de metapadrões. Na classe `ItemAlugado` o método `calcula_diaria()` é o método hook, pois deve ser preenchido de acordo com um calculo específico para um determinado item alugado, por exemplo, um automóvel; o método `retorna_nome_item()` é um método hook que pode ser substituído. O método `imprime_fatura()` representa o método template.

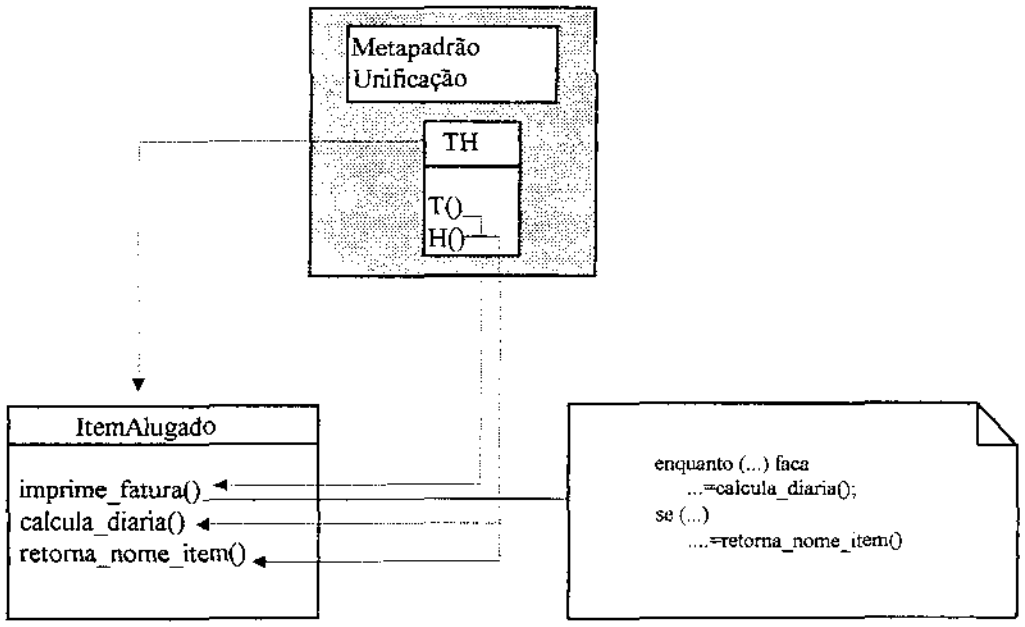



Figura 2.11: Método Template e Método Hook

Apesar dos métodos template e hook estarem encapsulados na mesma classe na Figura 2.11, nada impede dos mesmos estarem implementados em classes separadas. A classe que implementa os métodos hook é denominada classe hook (H) e a classe que implementa o método template é denominada classe template (T).

Além disso, metapadrões podem ser classificados em sete conjunto distintos[40] ilustrados na Figura 2.12. A Figura 2.12a) ilustra o Metapadrão unificação que encapsula métodos templates e hooks em uma mesma classe. A Figura 2.12b) ilustra o metapadrão conexão 1:1 onde um objeto de uma classe template refere-se a exatamente um objeto da classe hook. O metapadrão conexão 1:N, representado na Figura 2.12c), é usado para indicar que um objeto da classe template refere-se a qualquer número de objetos da classe hook. A Figura 2.12d) representa o metapadrão conexão recursiva 1:1, onde um objeto da classe template refere-se exatamente a um objeto da classe hook e a classe template é descendente da classe hook. Em uma versão modificada deste metapadrão, classe template e classe hook são unificadas resultando no metapadrão unificação recursiva 1:1 (Figura 2.12e). O metapadrão conexão recursiva 1:N, ilustrado na Figura 2.12f) indica que um objeto de uma classe template refere-se a qualquer número de objetos da classe hook e a classe template é descendente da classe hook. Em uma versão modificada, classes templates e hooks são unificadas resultando no metapadrão unificação recursiva 1:N (Figura 2.12g).


Figura 2.12 a) Metapadrão Unificação

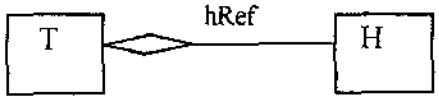



Figura 2. 12b) Conexão 1:1

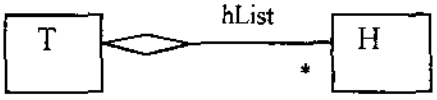



Figura 2.12 c) Conexão 1:N

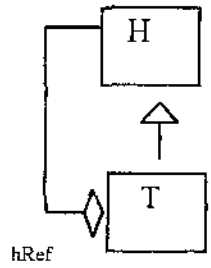



Figura 2. 12d)
Conexão Recursiva

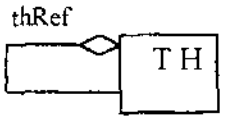



Figura 2. 12e) Unificação
Recursiva 1:1

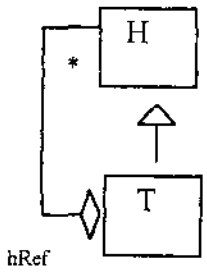



Figura 2.12 f)
Conexão Recursiva

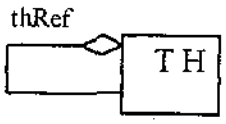



Figura 2.12 g) Unificação
Recursiva 1:N

Figura 2.12: Classificação de Metapadrões

## 2.3 Metodologias Orientadas a Objetos

Metodologia é um conjunto de técnicas que permite alcançar um objetivo específico. Uma metodologia para desenvolvimento de software é um conjunto de procedimentos para obter requisitos, analisar e projetar um sistema. É uma maneira de estabelecer uma padronização e assegurar uma análise rigorosa do problema em questão. A idéia de se adotar uma metodologia para desenvolvimento de sistemas começou com a análise estrutura. Entretanto, com o desenvolvimento da programação orientada a objetos, o enfoque do projeto de sistemas transferiu-se de funções para dados, levando à necessidade de novas metodologias.

Assim, novas metodologias orientadas a objeto surgiram, permitindo que conceitos como polimorfismo, encapsulamento e herança fossem considerados desde a análise até o projeto do software. Existem na literatura muitas metodologias orientadas a objetos. Para o desenvolvimento deste trabalho adotamos a metodologia UML[], uma vez que esta tende a se tornar a metodologia padrão para desenvolvimento de softwares.

A UML( Unified Modeling Language) resultou da unificação de três metodologias a saber: Booch[8], OMT[47] e OOSE, com o objetivo de eliminar elementos não utilizados na prática, adicionar elementos de outros métodos e criar novos elementos ainda não disponíveis. O resultado é a proposta de uma linguagem de modelagem simples que pretende ser universal no sentido de permitir a modelagem de aplicações de propósitos variados, uma vez que ela se destina inclusive a modelagem de sistemas de tempo real, sistemas distribuídos e sistemas concorrentes.

A UML define um conjunto central de diagramas, usados em diversas fases de análise ou desenvolvimento de software. Dentre estes diagramas podemos citar:

- Diagrama de Casos de Uso, usado para delimitar o sistema e definir sua funcionalidade
- Diagramas da Estrutura Estática: este diagrama inclui
    - Diagrama de Classes, que representa a estrutura estática do sistema através da descrição de classes, objetos, módulos, etc e seus relacionamentos;
    - Diagrama de Objetos, que representa uma instância de um diagrama de classe em um dado momento.
- Diagramas de Comportamento
    - Diagrama de Estados descreve o comportamento temporal de um objeto em resposta às interações com outros objetos do sistema
    - Diagrama de Seqüência modela cenários
    - Diagrama de Colaboração mostra a seqüência de mensagens envolvida em uma operação.
- Diagrama de Implementação
    - Diagrama de Componentes descreve o projeto físico de um sistema a nível de software e hardware
    - Diagrama de Disposições que especifica a plataforma na qual o sistema executa.

A UML também criou novos conceitos para soluções até então não disponíveis. Dentre estes conceitos podemos citar:

- Estereótipos, elementos que estendem a semântica da UML. Exemplo: evento, exceções, metaclasse etc.
- Restrições, semântica para condições predefinidas ou definidas pelo usuário.
- Threads e processos;
- Distribuição e concorrência;
- Padrões;
- Definição clara entre classe, tipo e instância;
- Refinamento;
- Interfaces e componentes;

O núcleo da UML é o diagrama de classe, que descreve a estrutura estática do sistema e seus relacionamentos. Os principais componentes do diagrama de classes são: classes, objetos, relacionamentos, estereótipos e módulos.

**Estereótipos**

Um estereótipo é essencialmente a metaclassificação de um elemento UML. Esta metaclassificação tem como objetivo distinguir os vários tipos de classes existentes em um modelo. Todo elemento na UML pode ter no máximo um estereótipo; este por sua vez, pode ser omitido se a semântica predefinida pela UML é suficiente para modelar o elemento. A sintaxe para um estereótipo é um texto entre os símbolos "<<" e ">>". No contexto desta dissertação, estereótipos foram usados para modelar um tipo de classe particular: a metaclasse, como mostra a figura abaixo:

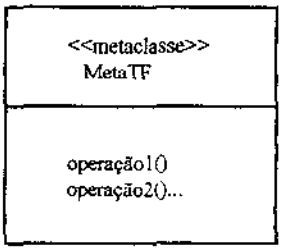



Figura 2.13: Exemplo de Estereótipos

**Módulos**

Outro conceito importante na UML é a noção de módulos. Módulos são subconjuntos de um modelo. Todos os elementos e diagramas da UML descritos anteriormente podem ser organizados em Módulos. Este conceito é interessante para designar agrupamentos lógicos, casos de uso e grupos de processadores. A figura abaixo representa um exemplo de Módulos e suas dependências:

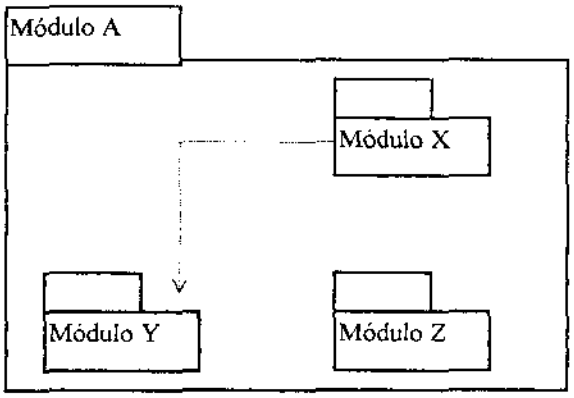



Figura 2.14: Exemplo de Módulos e suas Dependências

## 2.4 Resumo

Neste capítulo revisamos conceitos como encapsulamento, classe, herança e introduzimos o conceito de delegação, metaclasse, reflexão computacional, frameworks, e padrões de projeto. Apresentamos a notação adotada pela UML para a representação de dos conceitos provenientes do modelo de objeto e introduzimos um dicionário de terminologias orientadas a objeto empregadas nesta dissertação, uma vez que não há um acordo geral sobre várias definições aqui apresentadas. Esperamos assim, termos escolhido um conjunto consistente e claro de definições e contribuído de certa forma para o esclarecimento de alguns conceitos mais importantes no paradigma de orientação a objetos.

# Capítulo 3

# Fundamentos de Tolerância a Falhas

Sistemas computacionais são por natureza complexos, compostos por um grande número de subsistemas e componentes inter-relacionados: conseqüentemente, é provável que tais sistemas contenham falhas de software ou hardware. Diante deste fato, disponibilidade e confiabilidade tornam-se requisitos fundamentais para a construção de software. Disponibilidade assegura que o software está pronto para ser usado sempre que necessário. Confiabilidade assegura que o sistema é capaz de fornecer o serviço desejado e, segundo Lee e Anderson[26], é um requisito mensurável caracterizado por uma função R(t) que expressa a probabilidade que o sistema atenderá à sua especificação durante o período de tempo t.

Existem duas abordagens principais para a obtenção de sistemas confiáveis. A primeira abordagem, denominada **prevenção de falhas**, assegura que se durante a fase de implementação, o sistema for projetado visando diminuir falhas potenciais e durante a fase de teste, os erros que ainda permaneceram no sistema foram eliminados, o sistema está livre de falhas e pronto para ser usado. A segunda abordagem, denominada, **tolerância a falhas** acredita que mesmo um sistema já em uso não é perfeito e portanto algum mecanismo deve ser incorporado ao sistema de forma a tratar falhas que permaneceram durante seu desenvolvimento. Assim, podemos definir um sistema tolerante a falhas como aquele capaz de continuar a operar mesmo após a ocorrência de uma falha.

Uma discussão geral sobre projeto e implementação de sistemas tolerantes a falhas deve levar em consideração três fatores: (i) é necessário identificar os princípios básicos por traz de todo sistema tolerante a falhas, princípios aplicados a todos níveis de um sistema de computação e não somente hardware; (ii) os mecanismos que apoiam e implementam técnicas baseadas nestes princípios devem ser investigados; (iii) é necessário um framework para dar apoio a uma abordagem para tolerar falhas coerente e bem estruturada, de forma que a complexidade adicional introduzida pelas técnicas de tolerância a falhas não comprometam a confiabilidade do sistema. Em particular, esta dissertação concentra-se em investigar estes três fatores, como veremos neste e nos próximos capítulos.

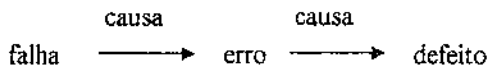
Este capítulo está assim dividido: a Seção 3.1 apresenta alguns conceitos básicos sobre tolerância a falhas. As Seções 3.2 e 3.3 discutem tolerância a falhas de hardware e software respectivamente. A Seção 3.4 apresenta a importância do modelo de objetos para sistemas tolerantes a falhas. A Seção 3.5 discute tolerância a falhas em sistemas distribuídos. A Seção 3.6 traz um breve resumo deste capítulo.

## 3.1 Conceitos Básicos

Esta Seção tem como objetivo definir alguns princípios básicos presentes em todos sistemas tolerantes a falhas.

### Falha, Erro e Defeito

Segundo a terminologia proposta por Lee e Anderson, um sistema é um conjunto de componentes que se interagem para executar uma determinada tarefa. Um sistema tolerante a falhas é aquele capaz de continuar a operar mesmo após a ocorrência de uma falha. Entretanto, uma questão importante a ser considerada é a distinção entre *falha*[1], erro e *defeito*[2]. A falha é um defeito no estado interno de um componente ou projeto que pode levar a um erro. O erro é um estado não válido do sistema que pode, por sua vez, levar a um defeito. Finalmente, o defeito de um sistema ocorre quando este desvia-se daquilo que lhe foi especificado. Resumidamente temos que:




Dois tipos de falhas devem ser considerados no desenvolvimento de técnicas de tolerância a falhas: falhas de hardware e falhas de software. Falhas de hardware são aquelas causadas por componentes físicos do computador e, apesar de não fazerem parte do escopo desta dissertação, são discutidas brevemente na Seção 3.2. Falhas de software são aquelas causadas por erros lógicos deixados no software durante seu

[1] Do inglês *fault*
[2] Do inglês *failure*

desenvolvimento e serão discutidas com mais detalhes na Seção 3.3. Ressaltamos que esta dissertação concentra-se particularmente em tolerância a falhas de software.

**Redundância**

Toda técnica de tolerância a falhas utiliza redundância de componentes críticos, ou seja, elementos redundantes no sistema, uma vez que eles seriam absolutamente desnecessários se pudéssemos garantir que o sistema não falha. Uma forma comum de se classificar redundância é separá-la em redundância de hardware e redundância de software. Redundância de hardware refere-se a replicação de componentes físicos(transistores, unidades de memória, discos, etc) usados em caso de falhas de hardware, enquanto que redundância de software está relacionada com a provisão de programas extras, usados em caso de falhas de software. Redundância pode ser implementada estática ou dinamicamente. Na redundância estática todas as réplicas ou versões executam concorrentemente de forma a mascarar a ocorrência de falhas. Na redundância dinâmica apenas um componente está ativo em um dado momento. Se uma falha é detectada durante sua execução, ele é substituído por outro componente. É importante salientar que esta dicotomia entre redundância estática e redundância dinâmica não é rígida e depende da estrutura imposta pelo sistema. Assim, visões diferentes de um mesmo sistema pode levar a diferentes classificações de redundância.

**Componente Ideal Tolerante a Falhas**

Um componente ideal tolerante a falhas consiste de um componente bem definido ao oferecer uma separação clara entre suas atividades normais e anormais. As atividades normais implementam os serviços normais do componente, enquanto que as atividades anormais ou de tratamento de exceções implementam as medidas para tolerar falhas que causam tais exceções. A estrutura de um componente tolerante a falhas é mostrada na Figura 3.1.

A Figura 3.1 mostra que exceções de interface são geradas na parte normal do componente; exceções de defeito e interface provenientes de componentes localizados em níveis inferiores do sistema invocam a parte de tratamento de exceções do componente; exceções locais também podem ocorrer e são tratadas no próprio componente. Se as exceções de níveis inferiores ou locais são tratadas com sucesso, o componente será capaz de proporcionar serviço normal; caso contrário, ele gerará um exceção de defeito para um nível superior do sistema. Componentes ideais tolerantes a falhas são implementados através de tratamento de exceções que será discutido na Seção 3.4.

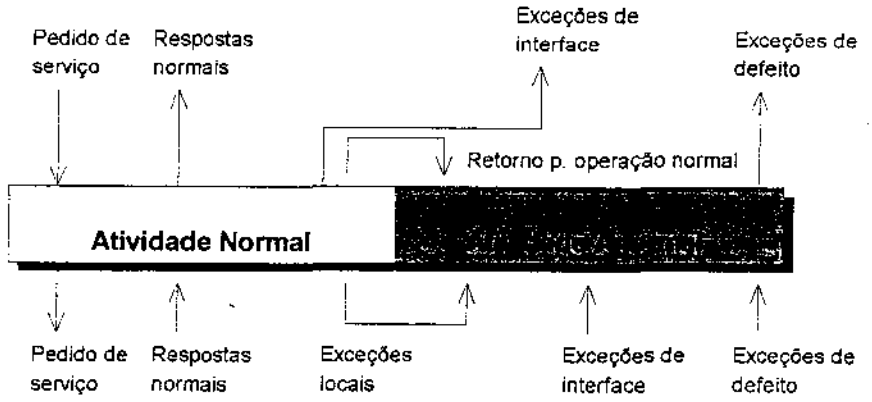



Figura 3.1: Componente Ideal Tolerante a Falhas

**Recuperação de Erros**

Como vimos anteriormente, uma falha em um componente pode levar a um estado incorreto do sistema que eventualmente pode causar um defeito. Para eliminar o sistema de um estado incorreto existem duas abordagens: recuperação de erro por avanço[3] e recuperação de erro por retrocesso[4][44]. Recuperação de erro por avanço procura recuperar-se de um estado incorreto corrigindo-se o erro que conduziu o sistema a este estado. Dessa forma, uma recuperação eficiente e específica pode ser implementada, desde que seja possível predizer qual a extensão e localização do erro precisamente. Como conseqüência, esta técnica possui as seguintes características:

(i) depende da dimensão do erro;
(ii) não é apropriada para erros não previstos;
(iii) é projetada para sistemas particulares;

Por outro lado, recuperação de erro com retrocesso procura recuperar-se de um estado incorreto restaurando o sistema para um estado anterior à ocorrência do erro, e portanto, supostamente correto. Restauração de estados é um método de recuperação possível para qualquer sistema, desde que a noção de estado seja inerente ao sistema. Como conseqüência, esta técnica possui as seguintes características:

(i) não depende da dimensão do erro;
(ii) fornece recuperação para erros arbitrários
(iii) é aplicável para vários sistemas ;
(iv) afeta o desempenho do sistema e necessita de recursos extra para armazenar informações.

[3] Do inglês *forward error recovery*.
[4] Do inglês *backward error recovery*.

É importante ressaltar que recuperação de erros por avanço e retrocesso são mecanismos complementares e portanto, um sistema tolerante a falha deve implementar ambas as técnicas.

**Tratamento de Exceções**

Um sistema constitui-se de componentes que produzem respostas a pedidos de serviços de outros componentes. Se um componente não é capaz de satisfazer um pedido, uma exceção é retornada para seu cliente. Assim, a resposta de um componente pode ser classificada em normal e anormal. Respostas anormais são freqüentemente chamadas exceções. Em cada nível do sistema, um componente ideal tolerante a falhas irá tratar uma resposta excepcional gerada por um componente num nível inferior, ou propagar uma exceção para um nível mais superior do sistema.

Exceções podem ser classificadas em duas categorias: exceções de interface[5], sinalizadas em resposta a um pedido que não corresponde à especificação da interface do componente; e exceções de defeito[6], sinalizadas para indicar a ocorrência de um erro no estado interno do componente.

Em geral, mecanismos para manipular exceções são necessários como forma de conectar ações de componentes do sistema que pertencem a níveis de abstrações diferentes. Tais mecanismos são chamados tratamento de exceções. Tratamento de exceções é uma técnica de tolerância a falhas de software que utiliza recuperação por avanço. As noções de exceções e de tratamento de exceções podem ser usadas para estabelecer uma infra-estrutura de apoio para a obtenção de tolerância a falhas de software. A idéia novamente se baseia em fornecer diversos componentes desenvolvidos de forma independente mas segundo a mesma especificação e um algoritmo de decisão para avaliar os resultados gerados. Executa-se uma das versões e submete-se o resultado ao teste de decisão. Caso o resultado não seja correto, uma exceção é gerada e o tratador associado a tal exceção é acionado. O código do tratador, por sua vez, contém chamadas para as demais versões. Em [15] encontramos uma definição formal para tratamento de exceções.

Apesar da maior parte dos trabalhos relacionados com tratamento de exceções ser dirigidos para linguagens imperativas (Ada e CLU) existe uma preocupação crescente de prover tais mecanismos também em linguagens orientadas a objetos. Um exemplo é o modelo para tratamento de exceções proposto em C++ e descrito no exemplo a seguir. Neste exemplo queremos coletar o menor elemento de um vetor e para isto contamos com os serviços de duas classes. A classe `ColetaRapida` ordena o vetor e então retorna o elemento da primeira posição. A classe `ColetaSegura` procura o menor elemento do vetor da forma mais simples: comparando-se todos seus elementos. A interface pública das duas classes é mostrada a seguir:

---

[5] Do inglês *interface exception*.
[6] Do inglês *failure exception*.

```
class ColetaRapida{
public:
    ColetaRapida(int* vetor,int estaOrdenado)
    void ordena();
    void asseguraordena();
    int minimo(); };

class ColetaSegura{
public:
    ColetaSegura(int* vetor);
    int minimo(); };
```

Uma vez que a classe `ColetaRapida` oferece um serviço mais eficiente porém mais complexo, ela será a primeira classe a receber o pedido do serviço. Entretanto, se a mesma não for capaz de fornecer o serviço corretamente, uma exceção será gerada e a classe `ColetaSegura` será invocada para fornecer o serviço desejado. Isto está mostrado no código abaixo:

```
class FalhaOrdenacao{} // Exceção
main()
  ...
  ColetaRapida cr(vetor, FALSE);
  r = cr.minimo();
  try{
      if (!ordenado())
        throw FalhaOrdenacao();
}
catch(FalhaOrdenacao) {
   ColetaSegura cs(vetor);
   r=cs.minimo();
}
```

Em C++ exceções podem ser declaradas como classes (FalhaOrdenacao) e suas instâncias são criadas sempre que uma exceção é gerada. Uma exceção é gerada dentro de um bloco **try.** Tratadores de exceção são declarados no final do bloco **try** usando-se a declaração **catch.** Para sinalizar uma exceção usamos a palavra **throw.** Sinalizar uma exceção é passar um objeto como argumento para um tratador. Este declara seu argumento como sendo de uma certa classe, podendo tratar também exceções objetos de suas subclasses. Portanto é possível aplicar o mecanismo de herança para construir uma hierarquia de exceções relacionadas e então tratá-las individualmente ou como um grupo. Mais informações sobre tratamento de exceções em C++ podem ser obtidas em [54] [25].

## 3.2 Falhas de Hardware

Os componentes de hardware constituem-se no ponto de partida para explorar a confiabilidade de sistemas por duas razões: a confiabilidade de qualquer software depende da confiabilidade dos componentes de hardware do sistema onde ele executa; e a falha de um componente físico pode, em princípio, ser identificada mais facilmente. Como conseqüência, a abordagem para tolerância a falhas tradicional, centrada apenas em falhas de hardware, levou ao desenvolvimento de vários modelos quantitativos para predizerem a confiabilidade de uma unidade de hardware. Nestes modelos, a noção de confiabilidade é expressa em termos probabilísticos em função do tempo e as medidas quantitativas baseiam-se nos seguintes detalhes:

(i) falhas ocorrem de forma independente, em componentes replicados independentes;
(ii) o comportamento de um componente de hardware pode ser conhecido através de observações de outros componentes similares;
(iii) o projeto do sistema não contem falhas.

A recuperação de tais falhas pode ser feita de várias formas. Uma técnica freqüentemente usada para obter consistência e tolerar falhas de hardware em ambientes não confiáveis trata-se de ações atômicas. Uma ação atômica consiste de uma transação envolvendo diversas operações em objetos compartilhados. Uma transação protege esses objetos compartilhados através da provisão das propriedades ACID - atomicidade, consistência, isolamento e durabilidade - para todas operações dentro da transação. Ações atômicas constituem-se numa forma útil para fatorar e organizar as interações entre componentes do sistema e se tornam particularmente eficientes quando podem ser enfatizadas por restrições impostas pela interação entre componentes ou fluxo de informações dentro dos sistema. Dentre alguns sistemas operacionais tolerantes a falhas de hardware podemos citar: ISIS[6], CORBA[60] e Arjuna[49].

## 3.3 Falhas de Software

Enquanto as técnicas para avaliar a confiabilidade de sistemas em relação a seus componentes físicos são bem estabelecidas, o mesmo sucesso não tem sido observado para técnicas que pretendem tratar falhas de software devido à diferença de natureza entre os dois tipos de falha. Falhas de software são causadas por erros lógicos e são difíceis de serem detectadas e tratadas. Como conseqüência, o comportamento de um software não pode ser conhecido pela simples observação de outro similar, como ocorre em componentes de hardware. Além disso, a definição de falha de software pode ser guiada também pela expectativa do usuário o que o torna um alvo móvel. Entretanto, a despeito destas dificuldades alguns modelos para avaliar e predizer a confiabilidade de software foram propostos. Em geral, estes modelos baseiam-se em redundância de projeto (ou diversidade de projeto) e não simplesmente replicação de programas. Redundância de projeto é uma técnica que propõe implementações diferentes para o

mesmo componentes. Cada uma dessas implementações é chamada de versão e são desenvolvidas com base numa especificação. Dessa forma, garante-se a probabilidade de minimizar a ocorrência de erros iguais. A diversidade de projeto implica em diversidade de soluções, uma vez que cada versão gerará seu próprio resultado. Para assegurar que o sistema retorne apenas uma solução, os resultados das versões são submetidos a um seletor ou algoritmo de decisão cuja função é verificar se uma solução particular é correta ou selecionar o resultado com maior probabilidade de estar correto a partir de um conjunto de soluções. Os resultados das versões são submetidos a um seletor (um algoritmo de decisão) que seleciona o resultado com maior probabilidade de estar correto. Duas técnicas tradicionais de tolerância a falhas de software são discutidas a seguir.

### 3.3.1 N-Versões

N-versões[5] é uma técnica de tolerância a falhas de software que utiliza redundância estática. Todas versões executam simultaneamente e um mecanismo de votação determina um único resultado de seleção de um conjunto ou subconjunto de todos os resultados das versões que executam em paralelo. O resultado aceito como correto (geralmente o resultado produzido pela maioria das versões) é então difundido para o resto do sistema. A Figura 3.2 exemplifica o funcionamento desta técnica:

A técnica de N-versões foi projetada sob uma perspectiva acadêmica. Na realidade, o desenvolvimento de três ou mais versões qualitativas e a um custo razoável constitui-se um grande desafio, o que torna seu uso um pouco questionável.

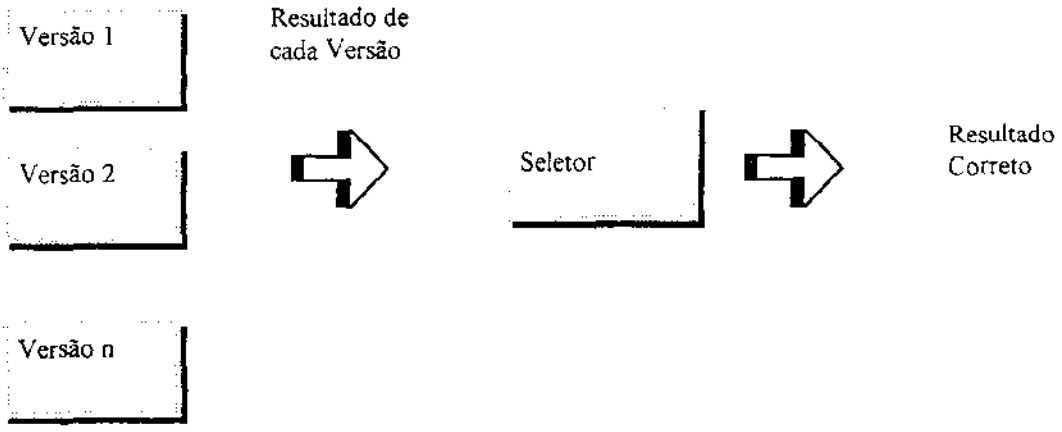



Figura 3.2: Funcionamento do N-versões

### 3.3.2 Bloco de Recuperação

O bloco de recuperação[59][43] é uma técnica de tolerância a falhas de software que utiliza redundância dinâmica. Neste esquema, as versões são nomeadas alternantes e o seletor é chamado de teste de aceitação. O teste de aceitação é aplicado seqüencialmente aos resultados obtidos pelos alternantes: se o primeiro alternante falha ao passar pelo teste de aceitação, o estado do sistema é restaurado usando-se técnicas de recuperação de erros por retrocesso, e o segundo alternante é executado; este processo continua até que todas as versões se esgotem ou algum resultado passe pelo teste de aceitação. A sintaxe para um bloco de recuperação é mostrada na Figura 3.3. O modelo de bloco de recuperação é bastante intuitivo e ajuda a formular uma estratégia para tolerância a falhas. A dificuldade está em desenvolver o teste de aceitação de forma a fornecer uma cobertura de erros adequada.

**assegure** *<teste de aceitação>*
**em** *<alternante 1>*
**senão em** *<alternante 2>*
...
**senão em** *<alternante n>*
**senao erro**

Figura 3.3 Sintaxe para o bloco de recuperação

## 3.4 Tolerância a Falhas de Software e o Modelo de Objetos

A Seção anterior discutiu algumas técnicas de tolerar falhas de software. Esta Seção apresenta uma abordagem orientada a objetos para implementar tais técnicas. A motivação é explorar as técnicas orientadas a objetos para projetar sistemas tolerantes a falhas, garantindo assim que a redundância inerente a estes mecanismos sejam incorporados de forma disciplinada, de maneira que o impacto na complexidade do sistema possa ser mantida sob controle. Além disso, devemos ressaltar que o uso de técnicas orientadas a objeto permitem a construção de componentes reutilizáveis provendo um framework para o desenvolvimento de sistemas tolerantes a falha. Em geral, a Seção anterior apresentou uma coleção de abstrações para construção de componentes tolerantes a falhas. Portanto, é necessário contarmos com um mecanismo poderoso para a construção destes componentes. Acreditamos que as técnicas orientadas a objetos permitem viabilizá-los através de mecanismos como herança, funções genéricas, polimorfismo e reflexão.

Entretanto, ao se considerar uma abordagem orientada a objetos para implementar tais componentes, encontramos diferentes níveis de abstração para incluir a redundância

de software necessária, a saber: diversidade de operação, diversidade de objeto, diversidade de classe e diversidade de metaclasse.

- Diversidade de operação: Neste caso, as versões são métodos de classe e desenvolvidas independentemente a partir de uma mesma especificação. As operações referentes a tolerância a falhas podem ser encapsuladas dentro da classe como operações privadas.
- Diversidade de objeto: A redundância é fornecida por dados presentes nas classes. Tolerância a falhas é então obtida instanciando-se um grupo de objetos, a partir da mesma classe, que se diferenciam nos dados internos e invocando-se a mesma operação sobre o grupo.
- Diversidade de classe: As versões são objetos e, portanto, dados e operações podem ser implementados independentemente a partir da mesma especificação de um tipo.
- Diversidade de Metaclasse: Neste caso, existem classes redundantes no meta-nível que podem ser usadas para construir objetos tolerantes a falha no nível de aplicação.

A primeira abordagem, diversidade de operações, é considerada uma solução baseada em objetos e não uma estratégia orientada a objetos. A abordagem baseada em diversidade de metaclasse necessita de algum mecanismo de delegação para selecionar a metaclasse que controlará o objeto. Além disso, esta estratégia depende do modelo de reflexão adotado pela linguagem, como veremos no capítulo 4. A abordagem baseada em diversidade de classe é a mais usada freqüentemente.

Nesta abordagem, especificações de serviços são representadas por classes abstratas e as variantes são instâncias de subclasses diferentes que conformam com a especificação da classe base abstrata. A construção de variantes diferentes, essenciais à diversidade de projeto, pode tornar-se uma alternativa cara. Uma vantagem de se usar esta abordagem para estruturar programas orientados a objetos é que ela permite a construção de variantes através da reutilização de componentes já existentes. O mecanismo de herança permite uma substituição seletiva de partes de um componente. Assim, uma subclasse pode ser vista como uma variante de sua classe base, herdando ou refinando partes de seu comportamento. A infra-estrutura de controle (BR e NV) e o teste de aceitação são métodos encapsulados no objeto da aplicação.

Por exemplo, considere a classe `Pilha` que define dois métodos virtuais: `empilha()` e `desempilha()`. `PilhaLista` e `PilhaVetor` são duas variantes da classe `Pilha`, implementadas como classes concretas derivadas a partir de `Pilha`. A classe `PilhaRobusta` também é derivada de `Pilha` e define a parte visível da abstração que será usada pelos clientes da classe `Pilha`. Isto quer dizer que os clientes da classe `Pilha` fazem seus pedidos para os objetos da classe `PilhaRobusta`. Estes redirecionam tais pedidos para os objetos das classes variantes, encapsulados nos objetos `PilhaRobusta` e portanto não visíveis para os clientes. O conjunto de classes forma um agregado e apesar de haver vários objetos, para o usuário existe apenas um único objeto `PilhaRobusta` que controla toda a operação do conjunto do agregado. Quando

um objeto `PilhaRobusta` é instanciado, os objetos `ObjPilhaLista` e `ObjPilhaVetor` são também criados internamente, como mostra o código abaixo:

```
class PilhaRobusta: public Pilha{
private:
        Pilha* ObjPilhaLista;
        Pilha* ObjPilhaVetor;
public:
        PilhaRobusta(){
            ObjPilhaLista = new PilhaLista;
             ObjPilhaVetor=new PilhaVetor; }
.... };
```

Outra vantagem do uso de diversidade de classes é a possibilidade de construção de frameworks, uma vez que estes se apoiam no conceito de reutilização sob a forma de classes abstratas que podem ser particularizadas para adequarem-se a aplicações específicas. A definição de frameworks é útil para a construção de sistemas tolerantes a falhas no sentido de promover não apenas reutilização de componentes como também transparência para programadores da aplicação, que apenas utilizam seus serviços sem se preocuparem como eles são implementados. Isto reduz a complexidade do software e conseqüentemente a possibilidade de falhas.

Acreditamos assim que, sob a ótica de tolerância a falhas, a noção de objeto, sua classificação em classes e a possibilidade de construções de frameworks, inerentes ao modelo de objetos, são importantes e adequados para desenvolver aplicações dentro deste domínio.

## 3.5 Tolerância a Falhas de Hardware em Sistemas Distribuídos

No escopo desta dissertação, uma aplicação distribuída é uma coleção de componentes de softwares distribuídos comunicando-se através de mensagens. Estas aplicações caracterizam-se pela descentralização de controle entre os componentes de software que executam independente e em cooperação, trocando informações e oferecendo serviços a outras aplicações. A implementação de aplicações tolerantes a falhas distribuídas depende muito do ambiente em que a aplicação executa. Como conseqüência, uma aplicação tolerante a falhas distribuída deve prover mecanismos para tolerar falhas de hardware .

Duas técnicas de tolerância a falhas de hardware para ambientes distribuídos são freqüentemente: replicação de componentes de software (réplicas dos mesmos softwares em vários nós da rede para aumentar sua disponibilidade) e pontos de recuperação em armazenamento estável. Em [18] existe uma descrição de como estas técnicas podem ser

implementadas segundo três abordagens diferentes: a nível de sistema, a nível de bibliotecas e a nível de herança. Na abordagem a nível de sistema, o sistema operacional onde a aplicação executa oferece os mecanismos para tolerar falhas de hardware. Como exemplo podemos citar as estratégias de replicação (ativa, passiva e semi-ativa) implementadas em Delta-4[39]. A vantagem desta abordagem é que na maior parte dos casos ela é transparente para a aplicação. A principal desvantagem é que ela não é flexível, uma vez que a aplicação torna-se dependente de um sistema específico. A abordagem a nível de biblioteca baseia-se no uso de bibliotecas pré-definidas e primitivas na forma de *tool-kit* para implementar tolerância a falhas. Um exemplo notável do uso desta abordagem pode ser encontrado no ISIS. As abordagem anteriores não se comprometem com propriedades particulares de linguagens de programação, uma vez que elas se baseiam em mecanismos oferecidos pelo sistema ou por um ambiente específico. Ao contrário, a abordagem a nível de herança usa as vantagens da programação orientada a objetos para implementar os mecanismos de tolerância a falhas. Técnicas de tolerância a falhas de hardware estão presentes em classes pré-definidas. Características não funcionais como persistência e recuperação são herdadas de tais classes. Essa solução foi usada com sucesso no Arjuna. O Arjuna é um sistema de programação orientado a objetos, implementado em C++, que provê um conjunto de classes que auxiliam a construção de aplicações distribuídas tolerantes a falhas, estruturadas como ações atômicas operando sobre objetos persistentes. Em Arjuna, uma classe pode ser declarada como recuperável. Isto significa que suas instâncias podem se recuperar de erros de hardware, uma vez que algumas definições são dadas pelo projetista da classe (definições de funções virtuais e sobrecarga de funções). Uma classe recuperável é derivada da classe `StateManager`, responsável pela provisão de persistência e mecanismos de recuperação. O programador da aplicação deve definir as funções virtuais `save_state()` e `restore_state()` para a classe recuperável para salvar/restaurar os estados dos objetos que são armazenados em um repositório de objetos estável. Assim, quando uma falha ocorre, uma nova computação é iniciada a partir da informação armazenada no repositório de objetos.

Uma vez que consideramos o ambiente de suporte das aplicações distribuídas tolerantes a falhas de hardware, podemos abstraí-lo, centralizando apenas na aplicação. Então tolerância a falhas de software pode ser obtida aplicando-se as técnicas descritas na Seção 3.3 deste capítulo. O estudo de caso apresentado na Seção 3.4 foi implementado para um ambiente distribuído em [38]. O experimento usou o ambiente Arjuna para tolerar falhas de hardware e estruturas de n-versões e blocos de recuperação para tolerar falhas de software. Mais informações sobre sistemas tolerantes a falhas em sistemas distribuídos podem ser encontradas em [16].

## 3.6 Resumo

Neste capítulo vimos que aplicações de software modernas devem atender requisitos de confiabilidade, disponibilidade, tolerância a falhas, etc. Em particular, a provisão de tolerância a falhas está baseada na noção de redundância de componentes do sistema, tanto para a detecção de erros quanto para sua recuperação. Para tolerar falhas de software precisamos de redundância de software ou diversidade de projeto. Duas técnicas clássicas para tolerar falhas de software são o bloco de recuperação e n-versões. Uma generalização simples desses esquemas é a seguinte[4]: (i) diversas variantes são implementadas para cada componente de software; (ii) cada variante é desenvolvida independentemente, mas devem seguir a mesma especificação de interface; (iii) uma infra-estrutura de controle executa as variantes; e (iv) os resultados são avaliados por um árbitro.

Por outro lado, a incorporação de redundância implica em custos maiores de desenvolvimento do sistema. Como software é o componente mais crítico de um sistema, ele tende a ser o gargalo de confiabilidade. Como conseqüência, o uso de redundância de software deve ser feito de modo estruturado, cuidadoso e disciplinado.

Programação orientada a objetos está sendo considerada como um modelo de programação efetivo para a produção de software de melhor qualidade. Seus benefícios incluem: abstração de dados, encapsulamento, modularidade hierarquia, polimorfismo, e reutilização de componentes. Em particular, mostramos na Seção 3.4 que o uso de técnicas orientadas a objetos para implementar mecanismos de tolerância a falhas facilita o controle da complexidade do sistema porque elas promovem uma melhor estruturação dos componentes do sistema. Além disso, mostramos que as técnicas orientadas a objeto facilitam a noção de interfaces abstratas de serviços que permitem diferentes implementações. Isto pode ser obtido usando-se uma classe abstrata para descrevera a interface desejada e classes derivadas para cada tipo de implementação. A idéia de tipos abstratos de dados com diferentes implementações associados à noção de um objeto externo gerenciando estas implementações é a forma principal para manter a complexidade do sistema.

Vimos também que para aplicações tolerantes a falhas distribuídas devemos considerar as falhas de hardware provenientes da existência da rede bem como as falhas de software da própria aplicação. Falhas de hardware podem ser mascaradas e tratadas por ambientes para programação distribuídas, como o Arjuna e ISIS. Uma vez tratadas as falhas de hardware, elas podem ser abstraídas. Então as falhas de software podem ser tratadas com técnicas de diversidade de projeto.

# Capítulo 4

# Reflexão Computacional

O conceito de reflexão aparece em diversas ciências como, filosofia, lingüistica, lógica e computação, geralmente com conotações e motivações diferentes. Por exemplo, no sentido de cognição humana, reflexão expressa a habilidade de meditar sobre idéias, ações, sentimentos e experiências. Em computação, reflexão surgiu em inteligência artificial como forma de resolver alguns problemas, como o fato de sistemas não serem capazes de se adaptar a novas situações. Para resolver tal problema, os sistemas passaram a implementar, além da própria aplicação a que se destinavam, um mecanismo que os permitia refletir sobre si mesmos. Essa idéia foi usada primeiramente por Brian Smith[51], que define a hipótese reflexiva da seguinte forma:

*"Assim como um processo computacional pode ser construído para refletir um mundo externo devido à presença de um interpretador que manipula formalmente representações deste mundo, também pode-se construir um processo computacional para refletir sobre si mesmo, devido à presença de um interpretador que manipula formalmente representações de suas próprias operações e estruturas."*

Entretanto, para que o mecanismo de reflexão seja implementado de forma flexível e modular é preciso definir uma arquitetura reflexiva, onde os requisitos relacionados com reflexão são tratados com um domínio válido, fornecendo-lhe a mesma modularidade apresentada pela aplicação e provendo-lhe uma forma de representação. Isto é o que veremos na próxima Seção. O restante deste capítulo está dividido da seguinte forma: a Seção 4.2 apresenta três modelos de reflexão computacional; a Seção 4.3 discute como reflexão pode ser usado para obter transparência em aplicações; a Seção 4.4 apresenta alguns trabalhos relacionados com reflexão a nível de sistemas, linguagens de programação e tolerância a falhas. Finalmente, a Seção 4.5 traz uma breve conclusão deste capítulo.

## 4.1 Arquitetura Reflexiva

De maneira geral, uma aplicação depende de dois tipos distintos de requisito: requisitos funcionais, associados ao propósito da aplicação *e* requisito administrativos, relacionados com a infra-estrutura de apoio do sistema para a realização deste propósito. Com domínios de atuação tão diferentes era de se esperar que uma aplicação bem modularizada apresentasse uma separação nítida entre estes requisitos. O uso de técnicas orientadas a objeto, como encapsulamento e definições de interface externas melhoram a modularidade da aplicação mas não são suficientes para garantir tal separação. Esta modularidade, entretanto, é facilmente obtida com o uso de reflexão, uma vez que esta não se destina ao domínio externo(funcional) da aplicação, mas sim à sua organização interna e interface para o mundo exterior. Isto só é possível devido a utilização de uma arquitetura multi-níveis chamada arquitetura reflexiva. Uma arquitetura reflexiva é aquela que dá apoio ao conceito de reflexão. Ela se constitui basicamente em dois níveis: **nível base ou nível de objeto**, associado ao domínio da aplicação e ideal para implementar requisitos funcionais, e o **meta-nível ou nível de metaobjetos**, relacionado com a solução de problemas e armazenamento de informação sobre o nível base, e portanto, ideal para a implementação de requisitos administrativos. Uma conseqüência desta arquitetura é que reflexão torna-se útil para estruturar atividades administrativas da aplicação, uma vez que estas podem ser mudadas de acordo com novas necessidades, sem no entanto interferir na computação do mundo externo. Um bom exemplo de uma arquitetura reflexiva e as vantagens provenientes de seu uso pode ser encontrado em [57]

.

No Capítulo 2 vimos que um framework consiste de várias classes abstratas altamente dependentes entre si. Em algumas circunstâncias, tais dependências tornam-se muito complexas e, como conseqüência, ao estender o framework, o programador deve tomar cuidado com o protocolo que estabelece as restrições necessárias para manter a coordenação das funcionalidades presentes no mecanismo. Arquiteturas reflexivas possibilitam a simplificação dos protocolos presentes no framework, separando-se as definições referentes às classes das definições das interdependências entre elas (Figura 4.1). Dessa forma, o comportamento de objetos condicionados a uma única classe são descritos normalmente pelas classes bases mas o comportamento de objetos condicionados a várias classes é descrito pelas metaclasses no meta-nível. Ao descrever

as interdependências de classes no meta-nível, as interdependências mínimas necessárias para o funcionamento do framework são preservadas, caso novas subclasses sejam estendidas pelo programador. Como resultado, o protocolo implementado pelo programador da aplicação é reduzido, uma vez que as interdependências são reutilizadas. A figura 4.1 exemplifica uma arquitetura reflexiva.

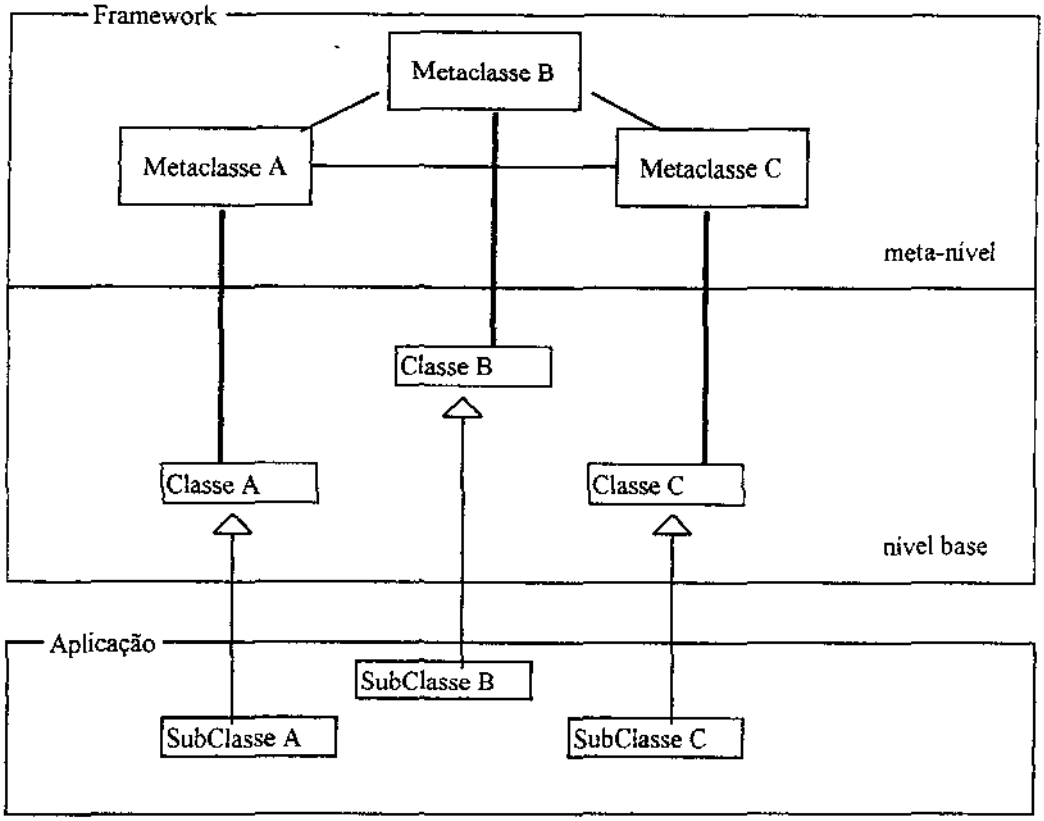



Figura 4.1: Arquitetura Reflexiva Implementando um Framework

Nas arquiteturas reflexivas, a interface entre o meta-nível e o nível base é nitidamente separada pelo PMO (Protocolo de Metaobjeto da linguagem de programação.

## 4.2 Modelos de Reflexão

Segundo Ferber[19], reflexão computacional pode ser implementado segundo três modelos principais: **modelo de metaclasse, modelo de metaobjetos** e **modelo de metacomunicação**. No modelo de metaclasse, todas instâncias de uma classe são controladas por um único metaobjeto, como mostra a Figura 4.2-a . Como conseqüência, este modelo é menos flexível que os demais. No modelo de metaobjeto, cada objeto pode ser controlado por um metaobjeto correspondente, como mostra a Figura 4.2-b. Esta abordagem é mais flexível que a abordagem anterior, uma vez que ela especializa o comportamento do metaobjeto de acordo com o objeto individual. No modelo de metacomunicação mensagens são objetos de uma classe pré-definida no sistema, por

exemplo Message, e tal que toda invocação de método torna-se uma instância desta classe. A classe Message é responsável por interpretar mensagens. Assim, quando um objeto invoca um método, uma instância da classe Message é criada para interpretar a mensagem e uma mensagem Send() é enviada para esta instância, como mostra a Figura 4.2-c. Este modelo é mais flexível que os anteriores porém tem a desvantagem de estar sempre operando no meta-nível uma vez que toda mensagem deve ser materializada[1].

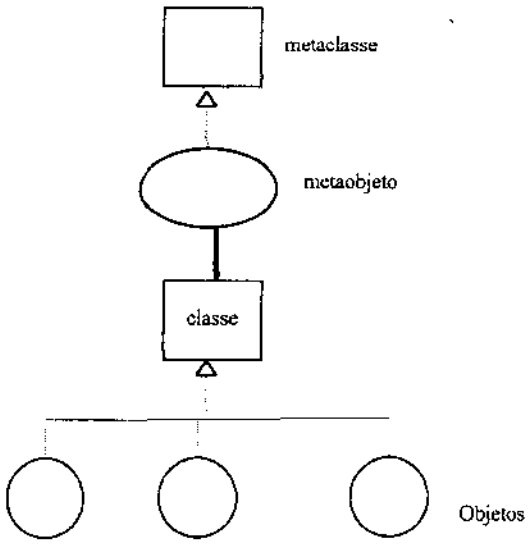



Figura 4.2-a Modelo de metaclasse

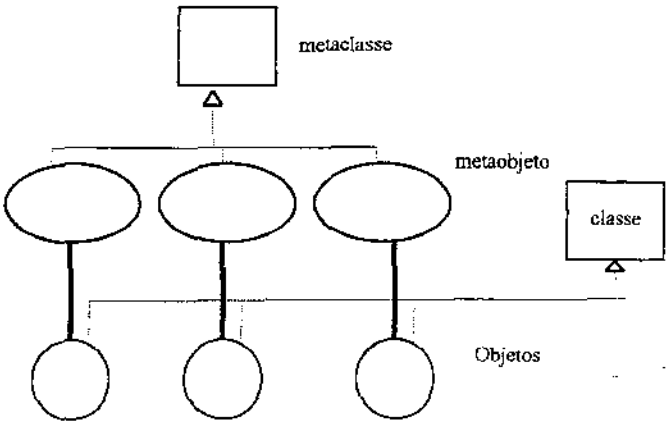



Figura 4.2-b Modelo de metaobjeto

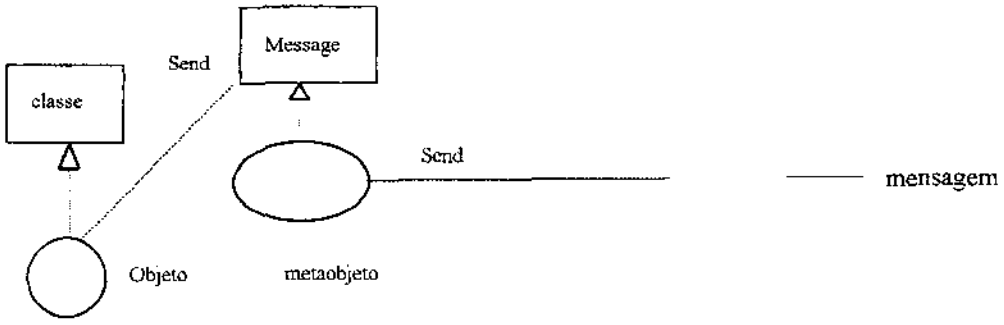



Figura 4.2-c Modelo de metacomunicação

[1] Do inglês *reified*

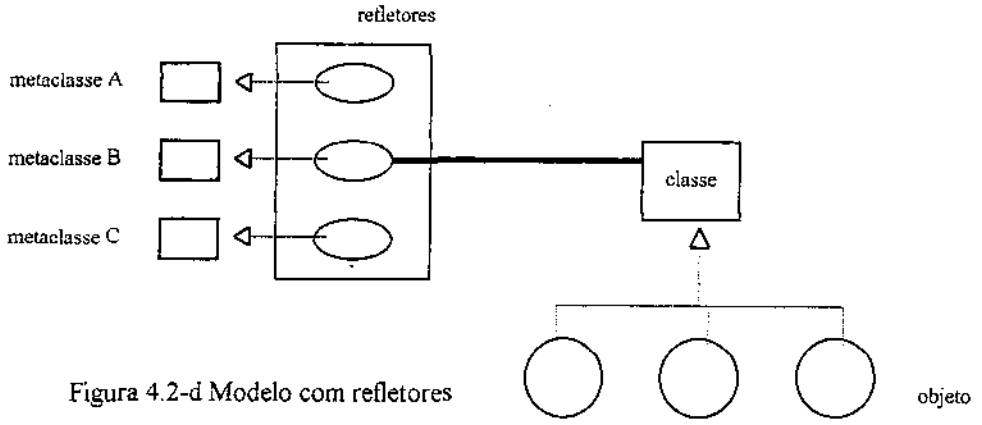



Figura 4.2-d Modelo com refletores

Figura 4.2: Modelos de Reflexão

Outro modelo adotado pela Sony no projeto do Apertos[61] trata-se **do modelo com refletores**, Figura 4.2-d. Neste modelo, um objeto pode estar associado a mais de um metaobjeto. Para guardar a informação de quais metaobjetos estão associados a um objeto, usamos objetos chamados refletores. Assim, refletores armazenam a associação "meta-de".

## 4.3 Um Novo Estilo de Programação

A arquitetura reflexiva compõe-se de meta-níveis, onde se encontram as ações a serem realizadas sobre um sistema alvo localizado no nível base. Esta arquitetura introduz um novo estilo de programação onde metaobjetos guardam informações sobre um objeto do nível base, dito seu referente. Para cada objeto **O** pode existir um metaobjeto $\mathbf{M_O}$ que representa os aspectos estruturais e comportamentais de **O** . Cada mensagem **s** enviada para um objeto **O** é interceptada e dirigida ao seu metaobjeto $\mathbf{M_O}$ . O metaobjeto $\mathbf{M_O}$ encarrega-se de sua execução, que é realizada no nível base controlada pelo meta-nível. O objeto emissor da mensagem **s** não toma conhecimento da computação reflexiva: ele envia a mensagem solicitando serviços de um objeto e recebe o resultado esperado, sem saber que a mensagem foi desviada para um metaobjeto. Essa noção é um modo interessante e conveniente de implementar transparentemente certas atividades administrativas do sistema, como por exemplo, tolerância a falhas, depuração de erros, otimização e políticas de segurança.

## 4.4 Programação Orientada a Objeto e a Abordagem Meta-nível

Programação orientada a objetos provê um meio particularmente interessante para construir sistemas meta-nível. Pode-se dizer que implementações orientadas a objeto fornecem uma representação de suas implementações uma vez que se baseiam em objetos que fornecem todos os métodos necessários para implementar o sistema. Além disso, programação orientada a objetos garante que o meta-nível será implementado de forma reutilizável. Entretanto, é importante ressaltar a separação dos benefícios de uma abordagem e a natureza do meio na qual ela está incorporada. E sendo assim, muitos benefícios atribuídos à programação orientada a objetos devem-se ao fato de escrever implementações abertas, ou seja, estruturar sua implementação de forma a permitir aos usuários acessá-la. Muitas técnicas orientadas a objetos podem ser revistas para uma abordagem meta-nível. Um exemplo notável disto é a divisão em camadas de protocolos de objetos que no contexto de uma abordagem meta-nível é usada para atingir dois objetivos: permitir aos usuários modificar implementações em diferentes níveis de alterações e permitir aos implementadores dividir a responsabilidade da implementação de comportamentos entre vários componentes meta-nível. Aspectos como polimorfismo e herança também são relevantes para a abordagem meta-nível, uma vez que clientes podem se beneficiar de bibliotecas meta-nível já disponíveis.

Outro aspecto a ressaltar é que a abordagem meta-nível enfatiza um aspecto pouco explorado pelas pesquisas sobre o paradigma de programação orientada a objetos. Até então, programação orientada a objetos era conhecida pelos benefícios provenientes da reutilização de código. Trabalhos em sistemas reflexivos mostraram que um outro grande benefício do modelo de objetos vem da possibilidade de se explorar aspectos de implementação do sistema. O desafio está em projetar arquiteturas que ofereçam estrutura adequada e oportunidade para especialização e evolução.

## 4.5 Usos de Reflexão Computacional

Reflexão computacional tem sido explorado em áreas como sistemas distribuídos, linguagens de programação, banco de dados e tolerância a falhas, com o objetivo de se obter sistemas mais flexíveis e modulares. Assim, As Seção 4.4.1 descreve alguns trabalhos relacionados com reflexão e sistemas distribuídos. A Seção 4.4.2 abrange alguns trabalhos relacionados com reflexão e linguagens de programação. A Seção 4.4.3 descreve alguns trabalhos com reflexão e sistemas tolerantes a falhas.

### 4.5.1 Reflexão em Sistemas Distribuídos

Em sistemas distribuídos, reflexão computacional é utilizado para gerenciamento de recursos de sistemas de forma transparente não intrusiva[53], geralmente implementados de formas "ad hoc" e pouco extensíveis em sistemas operacionais tradicionais. Alguns sistemas operacionais reflexivos são descritos a seguir. Um dos

exemplos mais notáveis de reflexão aplicado a nível de sistema é o projeto da Sony, Apertos, projetado a partir de um framework que introduz a separação entre objeto e metaobjeto. Um objeto é um "container"de informação e um metaobjeto define a semântica de seu comportamento. O modelo de reflexão usado no Apertos baseia-se em refletores. Neste modelo, objetos são controlados por um grupo de metaobjetos. Um grupo de metaobjetos, por sua vez, define um meta-espaço. Assim, a semântica de execução de um objeto é definida pelos metaobjetos que compõem o meta-espaço associado ao objeto. A informação de qual meta-espaço está associado a um objeto é armazenada em objetos chamados refletores. Como metaobjetos são também objetos, eles também podem ser associados a um meta-espaço, formando uma meta-hierarquia. A raiz desta meta-hierarquia é um metaobjeto chamado MetaCore que se comporta como um microkernel. A arquitetura reflexiva é usada para implementar migração como forma de tratar heterogeneidade de objetos. Assim, quando um objeto muda de propriedades ou de comportamento, ele deve migrar para o grupo de metaobjeto que fornece o comportamento desejado.

### 4.5.2 Reflexão em Linguagens de Programação

A técnica de programação orientada a objetos implica que o domínio da aplicação é estruturado em termos de objetos e não de procedimentos. Entretanto, o conceito de objeto não é o mesmo para todas linguagens orientadas a objeto. Assim sendo, segundo Masini et al.[33] podemos identificar pelo menos três famílias, cada qual enfatizando um ponto de vista particular sobre o conceito de objetos:

- Linguagens baseadas em classe: definem um objeto sob o ponto de vista estrutural, como um tipo de dados que encapsula uma estrutura e operações que podem ser aplicadas a esta estrutura;

- Linguagens baseadas em ``frames'': definem um objeto, sob o ponto de vista conceitual, como uma unidade de conhecimento. Um ``frame'' é um agente que descreve uma situação padrão ou um objeto. ``Frames'' podem também ser organizados em uma hierarquia de herança, mas ao contrário de classes, cada objeto é uma representação do ``frame'' que o originou e um gerador de ``frames'' mais especializados. Um ``frame'' é composto por ``slots'' que são descritos por facetas declarativas e procedurais. Facetas declarativas associam valores a "slots". Facetas procedurais são procedimentos ativados quando um "slot" é acessado. De maneira geral, linguagens baseadas em ``frames'' são aquelas que apresentam a tripla ("frame", "slot", faceta) como unidade básica de representação. Tais linguagens são projetadas principalmente para a representação do conhecimento.

- Linguagens baseadas em agentes: definem um objeto sob o ponto de vista de uma entidade ativa e autônoma, chamada agente. A comunicação entre agentes é feita através de mensagens assíncronas. O comportamento de um agente é definido por um ``script'' que descreve como um agente reage a eventos provocados por outros agentes. Agentes não são organizados hierarquicamente, ao contrário de classes: cada agente

pode delegar mensagens que não conseguem processar a outros agentes e desta forma, prover um mecanismo equivalente a herança (delegação). Devido ao seu aspecto dinâmico, linguagens baseadas neste modelo são particularmente desejáveis em programação concorrente.

Esta classificação é menos restritiva que a proposta por Wegner[63], onde as linguagens de programação são classificadas em baseadas em objetos, baseadas em classes e orientadas a objetos. Para Wegner, linguagens baseadas em objetos são aquelas que possuem apenas a noção de objetos. Igualmente, linguagens baseadas em classes são aquelas que possuem as noções de objetos e classes. Finalmente, linguagens orientada a objeto são aquelas que possuem as noções de objetos, classe e herança. Esta classificação, porém é restritiva pois não considera como linguagens orientadas a objetos algumas linguagens com outros mecanismos de compartilhamento de comportamento. Por isso, a classificação adotada nesta dissertação segue a divisão proposta por Masini.

Linguagens de programação diferenciam-se na forma como estas expressam suas representações internas, ou seus possíveis tipos de abstrações. Essencialmente, estas diferenças são traduzidas em quais aspectos internos do sistema computacional podem ser fornecidos implicitamente pela linguagem e quais devem ser codificados explicitamente. Estes aspectos fornecidos implicitamente são chamados absorvidos pela linguagem enquanto que os codificados explicitamente são chamados materializados. Uma linguagem eficiente pode então ser definida em termos de quantos detalhes ela pode absorver. Por outro lado, se uma linguagem pode absorver vários detalhes, seus programas tendem a ser menos gerais. Esta situação, por sua vez, conduz a uma aparente contradição entre generalidade e eficiência. Para resolver este problema, precisamos de um mecanismo onde os aspectos absorvidos pela linguagem possam tornar, em qualquer momento, explícitos, modificáveis e absorvidos novamente na implementação da linguagem. Portanto, necessitamos de um mecanismo que permita inspecionar e alterar as estruturas de implementação da linguagem. Linguagens que possuem tal mecanismo são chamadas linguagens de programação com implementação aberta, ou simplesmente "abertas".

Linguagens de programação tradicionais escondem do usuário sua implementação. Ao contrário, linguagens abertas, fornecem duas interfaces para seus clientes: a interface para as funcionalidades do sistema e a interface que revela os aspectos de como a interface anterior é implementada. Ambas interfaces, entretanto, podem ser facilmente reportadas para uma arquitetura reflexiva, resultando em uma interface do meta-nível que manipula a interface do nível base. Quando uma linguagem é implementada em uma arquitetura reflexiva chamam-na linguagem reflexiva.

Uma linguagem reflexiva consegue manipular suas entidades internas devido a presença de metaobjetos que geralmente obedecem um conjunto de regras denominado protocolo de metaobjeto ou PMO. Segundo Kiczales[24], o PMO é uma interface que permite eliminar a fronteira entre o usuário e o projetista da linguagem. Assim, os usuários podem estender ou modificar a semântica da própria linguagem, permitindo o

ajuste desta as suas necessidades específicas. Várias linguagens orientadas a objeto reflexiva já foram propostas, como veremos a seguir. As linguagens baseadas em frame foram as primeiras a introduzir a idéia de encapsular dados da aplicação com dados e métodos reflexivos. P. Maes introduziu uma arquitetura reflexiva na linguagem KRS, resultando na linguagem 3KRS[32]. Outro exemplo de linguagem reflexiva baseada em frame é RKL1[55], uma linguagem de programação lógica paralela e baseada em KL1. Os exemplos mais notáveis de linguagens reflexivas baseadas em classe são Smalltalk[22,29], OpenC++[11,12,13], CLOS[24,37]. O estudo de reflexão em linguagens baseadas em agentes levou ao desenvolvimento da linguagem Moostrap[36], onde o conceito de reflexão foi desenvolvido no contexto de Self. ABCL/R2[34,35,62] é outro exemplo de linguagem reflexiva baseada em agente. Na Seção 4.6 descrevemos o PMO destas linguagens.

### 4.5.3 Reflexão Computacional e Sistemas Tolerantes a Falhas

Tolerância a falhas é um exemplo típico onde requisitos administrativos podem ser aplicados, pois os mecanismos de tolerância a falhas estão mais ligados à máquina virtual que executa a aplicação do que à própria aplicação. Assim, reflexão pode ser usada para implementar diferentes estratégias de tolerância a falhas no meta-nível de modo transparente e não intrusivo, isto é, sem interferir na estrutura dos objetos monitorados localizados no nível base.

Alguns trabalhos já foram propostos para explorar reflexão computacional e orientação a objetos para tolerância a falhas de software. Xu et al.[58] apresenta uma idéia de uma arquitetura reflexiva para tolerância a falhas, mas resultados concretos não são mostrados. Zorzo et al.[65] apresenta uma avaliação de desempenho de uma aplicação reflexiva usando um mecanismo de injeção de falhas. Fabre et al.[18] mostra como reflexão pode ser usada para implementar protocolos de replicação em sistemas distribuídos, portanto, dando ênfase a tolerância a falhas de hardware. MOFT[27] - Metaobjetos para Tolerância a Falhas - apresenta um framework para a construção de aplicações tolerantes a falhas em uma arquitetura reflexiva. Neste trabalho encontramos exemplos de aplicabilidade em técnicas clássicas de tolerância a falhas de software. Uma evolução desta arquitetura está descrita em [28] para o desenvolvimento de aplicações distribuídas confiáveis.

## 4.6 Protocolo de Metaobjetos (PMO)

Nesta seção vamos focalizar um aspecto pouco usual das linguagens de programação: o protocolo de metaobjetos. Ele permite aos usuários customizar o comportamento e implementação da linguagem para suas necessidades específicas. Vamos analisar o escopo do PMO em algumas linguagens reflexivas. Porém, falar sobre o PMO obrigatoriamente exige uma mudança de ponto de vista: devemos mover da visão de programadores de aplicação em direção a implementadores de arquiteturas de linguagens. Esta arquitetura deve por sua vez permitir a transferência de alguns

"poderes" de implementadores para usuários. Explicamos e ilustramos a seguir os PMOs presentes em algumas linguagens de programação.

### 4.6.1 Open C++1.2

OpenC++ 1.2[11,12] é uma extensão de C++ que permite o uso de reflexão no acesso a atributos e na chamada de métodos. OpenC++ adota o modelo de metaobjetos, uma vez que cada objeto tem um único metaobjeto que pode controlar o acesso a variáveis e chamada de funções. Em OpenC++ existem dois tipos de objetos: reflexivos e não reflexivos. Um objeto reflexivo é controlado pelo seu metaobjeto que geralmente altera sua implementação. Um objeto não reflexivo é um objeto C++ normal. Métodos e atributos reflexivos são declarados através de diretivas ou pragmas no nível base, como mostra o exemplo abaixo:

```
1  class X{
2      public:
3       //MOP reflect:
4          x();
5          y();}
6   //MOP reflect class X : Y
7   class Y : public MetaObj{
8   public:
9     void Meta_MethodCall(Id m_id,
          ArgPac& args, ArgPac& reply){
10          Meta_HandleMethodCall(m_id,args,reply);}};
11  main() {
12       X x1;       // objeto normal
13       refl_X x2; }   // objeto reflexivo
```

Na linha 1 do exemplo acima, a classe X é declarada. Nas linhas 3 a 5, os métodos x() e y() são definidos como reflexivos através da diretiva **//MOP reflect:**. A linha 6 associa a classe X à metaclasse Y através da diretiva **//MOP reflect nome_classe : nome_metaclasse**. Esta, por sua vez, é definida nas linhas 7 a 10. Em OpenC++1.2, toda metaclasse é uma classe C++ que herda e redefine alguns métodos da classe `MetaObj`, como mostra a linha 7. Entre estes métodos destacamos: `Meta_MethodCall()`, linha 9, redefinido pela metaclasse para estender o método reflexivo e `Meta_HandleMethodCall()` linha 10, que realmente executa o método do nível base definido pelo programador. Quando o pré-processador encontra a classe `X` com métodos reflexivos, ele cria uma subclasse `refl_X` da classe `X`. Objetos normais são criados como instâncias de `X`, linha 12, e objetos reflexivos são criados como instâncias de `refl_X`, linha 13.

**Arquitetura do Meta-nível**: O PMO de OpenC++1.2 baseia-se na interceptação de mensagens. Quando um método reflexivo é chamado no nível base, sua execução é desviada para o meta-nível, que pode então manipulá-lo. Dois métodos são fundamentais neste processo:

- `Meta_MethodCall()`, usado para estender o método reflexivo;
- `Meta_HandleMethodCall()`, que realmente executa o método reflexivo.

A Figura 4.3 ilustra o mecanismo de interceptação de mensagens de OpenC++1.2. Quando um método reflexivo é invocado, esta chamada é interceptada pela metaclasse que passa a controlar sua execução. O método `Meta_MethodCall()` é então executado e, através do método `Meta_HandleMethodCall()` o método do nível base é executado. Em seguida, os resultados são retornados para o usuário. Isto implica que, quando um método reflexivo é chamado, o metaobjeto precisa saber qual é o método em questão, seus argumentos e onde armazenar seus resultados. Estas informações estão disponíveis em objetos "containers" da classe `ArgPac`, durante a invocação do método reflexivo. Esta classe comporta-se como uma pilha e implementa operações `push()` e `pop()` para cada tipo primitivo.

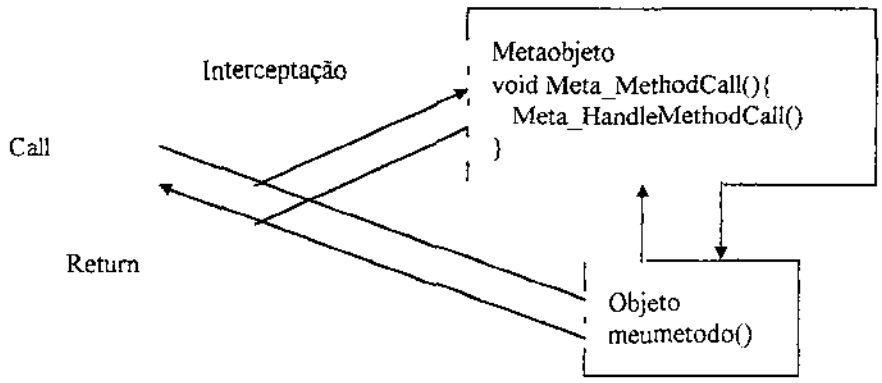



Figura 4.3: Funcionamento do protocolo de metaobjetos de OpenC++1.2

A hierarquia de metaclasses em OpenC++ 1.2 envolve basicamente cinco classes, como mostra a Figura 4.4:

- Classe `X`, cuja instância pode ser um objeto reflexivo;
- Classe `refl_X`, subclasse de `X` gerada pelo pré-processador OpenC++. O objeto reflexivo é uma instância direta dessa classe;
- Metaclasse de `X`;
- `MetaObj`, raiz da hierarquia de metaclasses;
- `NullMetaObj`, subclasse da classe `MetaObj` que implementa a semântica normal de C++ para chamada de métodos e acesso a atributos.

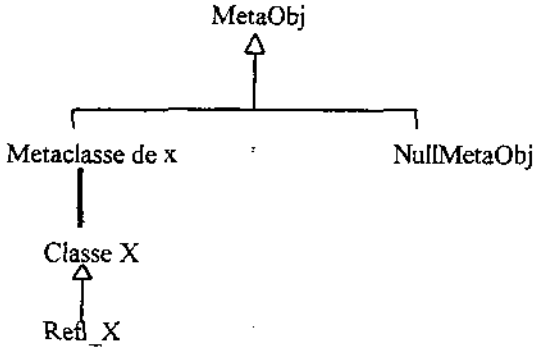



Figura 4.4: Hierarquia de metaclasses em OpenC++1.2

**Limitações**: Em OpenC++ 1.2 o MOP foi incorporado através de um pre-processador que implementa um mecanismo de interceptação de mensagens. Uma metaclasse é na verdade uma classe ordinária havendo apenas uma referência para a classe a qual está associada. Portanto não existe uma relação de instanciação entre elas. Isto limita a aplicação, no sentido que a metaclasse não tem acesso aos atributos definidos na classe. Para solucionar este problema, temos que romper o encapsulamento, fazendo com que tais atributos sejam passados como parâmetros para seus próprios métodos, a fim de que possam ser manipuladas pela metaclasse. Informações mais detalhadas sobre estes trabalhos e outros abrangendo reflexão computacional e tolerância a falhas são discutidos no Capítulo 7, na Seção de Trabalhos Relacionados.

### 4.6.2 OpenC++ 2.0

Como a versão 1.2, OpenC++ 2.0 é também baseada em C++, e portanto uma linguagem baseada em classe. As estruturas da linguagem são as mesmas de C++, exceto as que o programador pode acrescentar com a programação no meta-nível.

**Arquitetura do meta-nível**: O pré-processador OpenC++2.0 consiste de três estágios: pre-processador, tradutor de fonte OpenC++ para fonte C++, e compilador C++. O MOP de OpenC++ é uma interface que controla o tradutor no segundo estágio. Ele permite especificar como um código OpenC++ estendido é traduzido em código C++ regular. Para ligar um programa do nível base e um programa meta-nível, OpenC++ introduz uma nova sintaxe para declaração de metaclasse. A metaclasse default é Class, mas o programador pode mudá-la usando: metaclass **class-name: metaclass-name [ (meta-arguments)]**. Isto declara a metaclasse para uma classe. **meta-arguments** é uma seqüência de identificadores, nomes de tipos, literais e expressões C++, delimitadas por (). OpenC++2.0 também permite novas sintaxes estendidas desde que as mesmas sejam definidas pelo programa meta-nível. Algumas extensões possíveis são: novos modificadores de tipo, especificador de acesso, comando estilo while, for, entre outros. Metaobjetos denominados `Ptree` possuem várias operações já implementadas que

permitem ao programador obter o significado sintático do texto do programa, representado na árvore de parse, e também modificá-lo. Assim, dado um metaobjeto `Ptree`, pode-se saber se o texto representa uma operação, um comando while, uma classe, etc. Os metaobjetos `Class` fazem o papel do MOP. A classe de um metaobjeto é selecionada pela declaração metaclass no nível base. Também há funções que permitem fazer a manipulação da tradução de OpenC++ para C++, tais como `Ptree* TranslateClassName(Enviroment* env, Ptree* keyword, Ptree* name)`, `Ptree* TranslateBody(Enviroment* env, Ptree* body)`, que compõem o protocolo de tradução. Estas podem ser sobrecarregadas pelas metaclasses definidas pelo programador.

### 4.6.3 CLOS

CLOS é uma extensão orientada a objetos de Common Lisp, baseada nos conceitos de classes, funções genéricas, métodos e herança múltipla. A linguagem adota o modelo de metaclasse, uma vez que todos objetos da mesma classe compartilham o mesmo metaobjeto. A linguagem CLOS consiste de cinco conceitos básicos chamados de elementos de programa, a saber: classes, ``slots'', funções genéricas, métodos, combinação de métodos.

**Arquitetura do Meta-nível**: Em CLOS, todos objetos são instâncias de classes, e o comportamento destes objetos é definido por métodos associados a suas classes. CLOS é um sistema metacircular: seu comportamento é caracterizado por um conjunto de classes CLOS, instâncias e métodos, que constituem seus metaobjetos. A representação física de uma instância é controlada pela sua metaclasse, que determina as estruturas de armazenamento que são usadas. Ao estudarmos CLOS é conveniente separarmos seus aspectos estáticos de seus aspectos dinâmicos. A parte estática de CLOS é chamada de arquitetura do meta-nível e descreve os componentes do sistema e as estruturas dos blocos de construção da linguagem (classes, ``slots'', funções genéricas, métodos e combinação de ,métodos). A parte dinâmica é descrita em termos de protocolos que manipulam os blocos de construção para obter um determinado comportamento para a linguagem. Os protocolos descrevem as atividades principais de funções genéricas e explicam como elas devem ser invocadas para implementar os padrões de comportamento da linguagem. Extensões e modificações da parte dinâmica são portanto implementadas definindo-se novos métodos em funções genéricas já existentes.

**Limitações**: Os conceitos de orientação a objetos incorporados em CLOS tiveram que ser compatíveis com o padrão Common Lisp já existentes. Isto forçou alguns compromissos que provavelmente não estariam no projeto da linguagem se não fosse a necessidade de manter tal compatibilidade. Por exemplo, provavelmente não haveria distinção entre funções genéricas e funções ordinárias e métodos seriam encapsulados em classes. CLOS é uma ótima linguagem para desenvolver projetos rápidos e protótipos de sistemas. Infelizmente, é difícil mover-se destes protótipos para sistemas eficientes e bem depurados, parte devido à grande flexibilidade que o ambiente fornece, parte porque

classes CLOS não encapsulam funcionalidade e a combinação de métodos torna difícil entender exatamente que parte de código esta executando em um dado momento.

### 4.6.4 Smalltalk-80

Smalltalk é uma linguagem OO baseada em classe e com herança simples. Portanto cada objeto é uma instância de uma classe.

**Arquitetura do Meta-nível:** Smalltalk implementa o modelo de reflexão baseado em metaclasse, pois todos objetos estão associados a um único metaobjeto. Os componentes ou blocos de construção primários da linguagem são objetos do meta-nível, e portanto chamados metaobjetos. Este metaobjetos incluem classes e métodos em Smalltalk. A arquitetura do meta-nível é então construída a partir das classes que descrevem estes metaobjetos e suas hierarquias associadas. Em Smalltalk, metaclasses são geradas automaticamente pelo sistema. Por exemplo, considere uma classe `A`. Sua metaclasse, denotada por `A class`, é gerada automaticamente e o usuário tem acesso a ela através do envio da mensagem `class` para `A`. Metaclasses auxiliam na criação e iniciação de instâncias e também na iniciação de variáveis de classes. Por exemplo para combinar a criação de uma instância da classe `A` com sua iniciação explícita, um método de classe estende o método de classe padrão `new()` com uma chamada para um método de iniciação de instância. Metaclasses também podem ser usadas pra estender a estrutura de uma classe padrão. Estas metaclasses podem ser redefinidas pela adição de novas variáveis de instância para a definição default automaticamente provida pelo sistema. Entretanto, estas extensões são limitadas devido a correspondência 1-1 entre classe e metaclasse. O estudo dos protocolos associados com estas classes revela que todos os métodos que criam objetos, isto é, instâncias, classes e metaclasses, sempre fazem uso de dois métodos primitivos contidos em `Behavior`: `new()` e `new:`. O método `new()` definido em `Metaclass` sobrepõem o método `new()` definido em `Behavior` para estabelecer a correspondência 1-1 entre classes e sua metaclasse privada.

**Limitações**: As limitações deste modelo vêm da generalidade do problema que ele tenta solucionar (o estado de classes) e a solução limita que ele provê (metaclasses geradas automaticamente pelo sistema). Apesar de classes serem tratadas como objetos, seu protocolo de criação é inconsistente com o protocolo de criação de objetos. Uma classe é criada pelo envido de uma mensagem `subclass: ... :category:...` para sua superclasse enquanto uma instância terminal é criada pelo envio de uma mensagem new para sua classe. Além disso, este modelo é muito limitado já que o número de níveis de metaclasse é constante. Como classes não podem compartilhar a mesma metaclasse, mecanismos comuns implementados no meta-nível não podem ser extraídos e implementados por uma metaclasse compartilhada.

### 4.6.5 RKL1

RKL1 é uma linguagem de programação lógica paralela reflexiva baseada em KL1. KL1 foi projetada para representação do conhecimento sobre classes importantes de objetos do mundo real bem como os relacionamentos entre eles, afim de automatizar o entendimento de linguagem natural.

**A arquitetura do Meta-nível**: O meta-nível em RKL1 tem dois conceitos importantes: metacorpo, usado para descrever o código do meta-nível, e Grupo de metaprocesso (MPG), que realiza o gerenciamento dos recursos entre os múltiplos objetos Adequando-se cláusulas podemos aumentar o meta-interpretador. Entretanto, por razões de eficiência, apenas o corpo das cláusulas podem ser envolvidos nesta operação. Isto significa que a linguagem pode refletir apenas nas invocações de corpo de cláusulas. O programador insere a **sintaxe %reflect MetaModulo: Metacorpo(Parametros...)** antes da cláusula que terá sua execução controlada. É necessário informar os nomes do metamódulo e metacorpo. Quando um metacorpo é especificado numa cláusula do nível base, o corpo da cláusula deve ser executado sob um interpretador (executor) adequado. O novo executor invoca os objetos do meta-nível, que por sua vez invoca o objeto do nível base, ou outros objetos ao invés dos originais. Com relação ao modelo de reflexão, RKL1 adota uma variante do modelo de metaobjetos uma vez que cada objeto pode estar relacionado a mais de um metaobjeto (metacorpos). Entretanto, por questão de eficiência, esta associação só existe para objetos reflexivos, não afetando os objetos normais.

**Limitações:** Como dissemos anteriormente, RKL1 é uma linguagem lógica reflexiva, cujo objetivo é implementar alguns mecanismos de distribuição e paralelismo no meta-nível. Em ambientes distribuídos é indispensável a presença de mecanismos de gerenciamento de recursos, o que implica na necessidade de compartilhamento de informações no meta-nível. Entretanto, na arquitetura de RKL1 é difícil compartilhar tais informações uma vez que cada metacorpo, por questão de eficiência, executa individualmente.

### 4.6.6 3-KRS

3-KRS é uma extensão reflexiva de KRS, uma linguagem usada para a representação do conhecimento e relacionada com linguagens baseadas em frames.

**Arquitetura do Meta-nível:** 3-KRS segue o modelo de metaobjetos. Cada objeto na linguagem tem seu próprio metaobjeto. Este, por sua vez, tem uma referência para o objeto a ele associado, portanto não existe uma relação de instanciação entre eles. Todas entidades da linguagem (instâncias, métodos, "slots", mensagens e metaobjetos) são também objetos. Metaobjetos guardam informações sobre a implementação e

interpretação de objetos. Assim, o interpretador está dividido em blocos que representam como certos tipos de objetos são implementados.

### 4.6.7 Moostrap

Moostrap é uma linguagem reflexiva cujo projeto foi motivado pelo estudo de reflexão em linguagens baseadas em agentes. A idéia era desenvolver o conceito de reflexão no contexto de Self. Em Self, a estrutura de objetos pode ser consultada através de objetos chamados mirrors. Estes objetos são criados explicitamente para fornecer uma interface entre um protótipo e sua representação. Entretanto, Self não pode ser considerada reflexiva uma vez que mirrors não são objetos reflexivos reais, mas sim, um meio de acessar - e não modificar - informações sobre objetos através de primitivas da linguagem. Assim, os projetistas envolvidos neste trabalho optaram por desenvolver uma nova linguagem (Moostrap) ao invés de modificar o núcleo de Self.

**Arquitetura do Meta-nível:** Moostrap implementa o modelo de reflexão baseado em metaobjetos, uma vez que cada objeto está associado a um metaobjeto que mantém as regras de acesso a seus "slots". Mensagens são materializadas em duas fases: a busca do "slot" correspondente ao seletor(nome do método) e em seguida sua aplicação. Todo metaobjeto implementa um método chamado `lookup()` que resolve a busca de "slots". Este método tem como argumentos o objeto onde a mensagem será aplicada e seu seletor (`lookup(rec sel)`). Metaobjetos são descritos por outros metaobjetos. Esta recursão infinita é interrompida introduzindo-se metaobjetos circulares, ou seja, objetos que são seus próprios metaobjetos. Existe um metaobjeto circular básico, chamado basic-meta-object, o qual inicialmente, todos objetos do sistema estão associados. Este metaobjeto inicial define um método `lookup(rec sel)` que retorna o "slot" desejado ou gera um erro, caso não o encontre. A fase de busca do seletor é primeiramente delegada ao metaobjeto do objeto da aplicação. O resultado da fase de busca do seletor é um "slot" capaz de entender uma mensagem `apply()`. Entretanto, cada "slot" está associado a um objeto chamado slot-executant. Este objeto implementa um método `apply` tendo como argumentos um array composto do objeto em que a mensagem será aplicada e seus parâmetros. Sempre que uma mensagem (`aSlot selector arg1...argn`) é executada, ela é redefinida da seguinte forma: (`slot-executant(aSlot) selector aSlot array-of(arg1...argn)`). Existem três tipos de slot-executant, um para cada tipo de "slot":

- data-slot-executant, associado a "slots" de dados. A chamada de seu método `apply` retorna o conteúdo do "slot";
- method-slot-executant, associado a métodos. A chamada de seu método `apply` executa o método armazenado no "slot";
- assign-slot-executant, associado a "slots" de atribuição. A chamada de seu método `apply` atualiza o conteúdo do "slot".

**Limitações:** Moostrap é uma linguagem interpretada muito flexível, não só por usar um mecanismo de delegação, mas também devido ao projeto do seu meta-nível. A fase de aplicação para um "slot" pode ser modificada facilmente definindo-se um novo

slot-executant (contendo uma novo método `apply` e associando-o com o slot correspondente. Como isto pode ser feito para os três tipos de executants, a linguagem permite o uso de reflexão no acesso a variáveis (leitura e escrita) e chamadas de métodos. Uma desvantagem porém, é que não há meios de criar objetos não reflexivos, de maneira que toda invocação de métodos resultará em um desvio para o meta-nível, onde a mensagem será decomposta nas fases de busca e aplicação. Assim, o sistema estará quase sempre operando no meta-nível.

### 4.6.8 ABCL/R2

ABCL/R2 é baseada numa arquitetura de grupo híbrida, ou seja , que combina uma arquitetura individual e uma arquitetura de ``group-wide". A linguagem provê suporte a grupos de objetos heterogêneos e grupo de recursos compartilhados, metagrupos e torres reflexivas de grupo/indivíduo e objetos não materializáveis.

**Arquitetura do meta-nível:** ABCL/R2 coloca no meta-nível controle de recursos compartilhados encapsulados em grupos de objetos, tais como comunicação, armazenamento, etc. Tais recursos limitados têm representação no meta-nível como objetos, e são compartilhados no nível base num grupo. Ao mesmo tempo, cada objeto tem sua torre reflexiva, permitindo que meta-operações sejam adaptadas por objeto base. Objetos num grupo compartilham recursos que são representados como grupo de objetos do núcleo no meta-nível. Como resultado, há uma torre de metaobjetos (torre individual) para cada objeto, e a torre de meta-grupos (torre de grupo) para cada grupo.

A torre individual determina a estrutura do objeto, incluindo seu script (método). A torre de grupo determina o comportamento do grupo, incluindo a computação (avaliação) do script dos membros do grupo, e é formada pelo grupo do núcleo. A torre individual é criada por uma técnica chamada progressão dinâmica de reflexividade. Para manter a relação de causalidade, apenas um objeto pode estar ativo na torre em um dado instante. Os outros objetos serão objetos conceituais ou forwarding. Quando o avaliador executa uma instrução de envio de mensagem de um metaobjeto, ele tenta enviá-la para o metaobjeto do objeto destino no mesmo nível. Se os níveis dos metaobjetos são diferentes, a mensagem é redirecionada pela torre. Como este redirecionamento é uma forma simples de delegação, tais objetos podem ser implementados usando-se objetos leves, diminuindo o overhead de comunicação.

A torre de grupo é criada através de uma técnica chamada criação atrasada. Todo objeto, exceto o gerenciador de grupo, é criado com uma referência para o gerenciador de grupo do grupo do qual ele é membro. O gerenciador de grupo é criado inicialmente sem uma referência para seu gerenciador, ou seja, o gerenciador de grupo do seu meta-grupo. Este é criado depois, quando for referenciado. Durante a criação do gerenciador de grupo, os outros objetos do grupo do núcleo não estão criados. Isto só ocorre quando eles

são referenciados via gerenciador de grupo. Ao mesmo tempo, o gerenciador de grupo referencia seu próprio ``group ID" causando a criação do metagrupo.

Em ABCL/R2, usuários podem criar objetos que executam mais eficientemente comparados a objetos padrão(reflexivos), porem, com a perda das capacidades reflexivas.

### 4.6.9 Avaliação

O projeto de PMOs é ainda difícil, parte em função da ausência de metodologias bem formuladas para projetá-los, parte devido a necessidade de entendimento de questões importantes que afetam tanto usuários como implementadores. Enquanto projetistas tradicionais tomam todas as decisões de projeto e implementação, projetistas de PMOs estão envolvidos com a difícil tarefa de realocar algumas decisões em uma arquitetura em camadas que permita ao usuário tomarem suas próprias decisões de projeto e implementação. Apesar da dificuldade, alguns PMOs já foram propostos para linguagens reflexivas. Em geral, estas linguagens diferem-se na forma como metaobjetos são criados (sempre que o objeto é criado ou sob demanda), se a linguagem possui realmente um protocolo de metaobjetos, se a linguagem permite objetos reflexivos e não reflexivos e finalmente, como a hierarquia de classes e meta-hierarquia são diferenciadas. A tabela 4.1 mostra as principais características da arquitetura de algumas linguagens reflexivas. Mais informações podem ser obtidas em [50].

| | OpenC++ 1.2 | OpenC++ 2.0 | Smalltalk | CLOS | Moostrap | RKL1 | ABCL /R2 | 3KRS |
|---|---|---|---|---|---|---|---|---|
| modelo OO | classe | classe | classe | classe | agente | frame | agente | frame |
| modelo reflexão | metaobjeto | meta-classe | meta-classe | meta-classe | metaobjeto | var. metaobjeto | metaobjeto | metaobjeto |
| cardinalidade | 1-1 | N-1 | N-1 | 1-1 | 1-1 | 1-N | 1-1 | 1-1 |
| PMO | √ | √ | | √ | √ | √ | √ | √ |
| metaobjetos sob demanda | | | | | | | √ | |
| objetos n. reflexivos | √ | | | | | √ | √ | |

Tabela 4.1: Comparação entre algumas linguagens reflexivas

## 4.7 Resumo

Neste capítulo apresentamos reflexão computacional como uma técnica promissora para desenvolver aplicações complexas que necessitam de um alto grau de modularidade. Em particular, foi colocado que reflexão pode ser usada para implementar requisitos administrativos no meta-nível de modo transparente e não intrusivo, sem interferir na estrutura dos objetos monitorados pela aplicação. O uso de uma abordagem

reflexiva tem as seguintes vantagens: (i) separar ações pertinentes ao domínio da aplicação das ações administrativas e de controle da máquina virtual; (ii) promover a reutilização de objetos; (iii) permite que o programador da aplicação especifique as estratégias de controle que melhor se adeguem à aplicação e (vi) proporciona uma maior adaptabilidade destas estratégias ao sistema.

Também mostramos alguns trabalhos correlatos que aplicam reflexão em sistemas, linguagens de programação e tolerância a falhas. Alguns exemplos mais notáveis são Apertos, sistema desenvolvido pela Sony, CLOS, uma linguagem orientada a objetos reflexiva e MOFT, um framework reflexivo para o desenvolvimento de aplicações tolerantes a falhas de software.

# Capítulo 5
# FOORD: Um Framework Orientado a Objetos Reflexivo e Distribuído para Tolerância a Falhas

Neste capítulo é descrito o framework FOORD orientado a objetos reflexivo e distribuído para tolerância a falha. Esse framework possui as seguintes características:

1 - Suporte para tolerar falhas em aplicações orientadas a objetos;
2 - Utilização do ambiente Arjuna para tolerar falhas de hardware através do uso de ações atômicas;
3 - Implementação das técnicas de bloco de recuperação, n-versões e tratamento de exceções para tolerar falhas de software;
4 - Uso de uma arquitetura reflexiva para implementar os mecanismos acima de forma transparente e não intrusiva para a aplicação

O framework FOORD faz uso do modelo de objetos juntamente com a técnica de reflexão computacional para implementar mecanismos de tolerância a falhas de software. Através do apoio para reutilização de código fornecida pelo modelo de objetos, podemos diminuir o custo final do sistema produzido. Por outro lado, reflexão provê a manipulação das entidades internas da linguagem através de metaobjetos que obedecem um protocolo. No contexto desta dissertação, reflexão computacional será utilizada para implementar mecanismos de tolerância a falhas de software de forma transparente, separando a funcionalidade das técnicas de tolerância a falhas dos serviços fornecidos pela aplicação. O restante deste capítulo está dividido da seguinte forma: a Seção 5.1 discute os benefícios de uma arquitetura reflexiva para tolerância a falhas; a Seção 5.2 traz a descrição das classes que compõem o framework FOORD; a Seção 5.3 apresenta alguns aspectos de implementação do framework; a Seção 5.4 mostra um estudo de caso que exemplifica o uso do framework; as Seções 5.5 e 5.6 apresentam os pontos adaptáveis do framework e os passos para sua utilização respectivamente; a Seção 5.7 traz uma avaliação do framework proposto e a Seção 5.8 traz um resumo deste capítulo respectivamente.

## 5.1 Benefícios da Arquitetura Reflexiva para Tolerância a Falhas

A organização multi-níveis da arquitetura reflexiva apresenta diversas vantagens para o desenvolvimento de aplicações tolerantes a falhas, a saber:

1. Preservação de funcionalidades: o meta-nível é dedicado às atividades de detecção e recuperação de erros através da monitoração de objetos da aplicação enquanto que o que o nível base é responsável pelos requisitos funcionais (Capítulo 1) da aplicação.

2. Transparência na detecção de erros: A arquitetura reflexiva permite implementar diferentes estratégias de tolerância a falhas no meta-nível de modo transparente e não instrusivo.

3. Reutilização de componentes: a possibilidade de estender ou modificar o comportamento de objetos individuais da aplicação, controlando suas interações com outros objetos sem comprometer seu encapsulamento propicia diversas formas de reutilização de componentes. A arquitetura reflexiva permite identificar três componentes reutilizáveis:

- classes responsáveis por serviços não críticos da aplicação: estas classes não são afetadas pelo fato de participarem de uma aplicação tolerante a falhas;
- classes reflexivas: responsáveis por serviços críticos da aplicação e portanto tolerante a falhas. Esta classe é derivada para estabelecer a associação entre um objeto da aplicação e o seu metaobjeto. A classe original do objeto continua a existir, sendo usada para a instanciação de objetos não reflexivos.
- metaclasses: elas definem os mecanismos de tolerância a falhas e podem ser reutilizadas.

4. Objetos reflexivos tolerantes a falhas: numa aplicação geralmente podem coexistir objetos tolerantes e não tolerantes a falhas. O uso de reflexão permite diferenciar estes dois tipos de objetos da seguinte forma: objetos tolerantes a falha são objetos reflexivos, ou seja, estão associados a um metaobjeto enquanto que os objetos não tolerantes a falhas são objetos comum da aplicação. Objetos que compõem a aplicação interagem entre si para atender os serviços requisitados e os objetos considerados críticos são controlados pelos metaobjetos.

Dentre os quatro modelos de reflexão discutidos na Seção 4.2 do Capítulo 4, o modelo de metaobjetos é o mais apropriado para tolerância a falhas por permitir associar metaobjetos a objetos individuais da aplicação. Esta característica permite que objetos normais e objetos reflexivos, instanciados de uma mesma classe, estejam presentes na aplicação

Todas estas vantagens descritas acima, entretanto, dependem do PMO da linguagem adotada para implementar a arquitetura reflexiva do sistema, ou seja, o PMO adotado pela linguagem de programação pode ser mais ou menos adequado para implementar os mecanismos de tolerância a falhas[14]. Isto nos motivou a estudar os diferentes PMOs fornecidos por diversas linguagens de programação orientadas a objetos (Seção x do capítulo 4). Com base neste estudo escolhemos OpenC++1.2 para implementação do framework. Esta decisão se deve aos seguintes fatores:

1. O pré-processador é de domínio público;
2. Ele estende C++, uma linguagem bem conhecida;
3. Permite a diferenciação entre objetos reflexivos e não reflexivos para compor os objetos da aplicação;
4. Adota o modelo de metaobjetos, o mais apropriado para aplicações tolerantes a falhas, como discutido anteriormente.

## 5.2 Descrição das Classes do Framework FOORD

A Figura 5.1 representa uma macro visão do framework FOORD. Nesta figura o framework é representado por três módulos(cliente, o módulo FOORD e o módulo Arjuna) e suas dependências. O módulo FOORD, por sua vez, é formado por duas classes básicas (`MetaObj` e `MetaTF`) e o módulo Network.

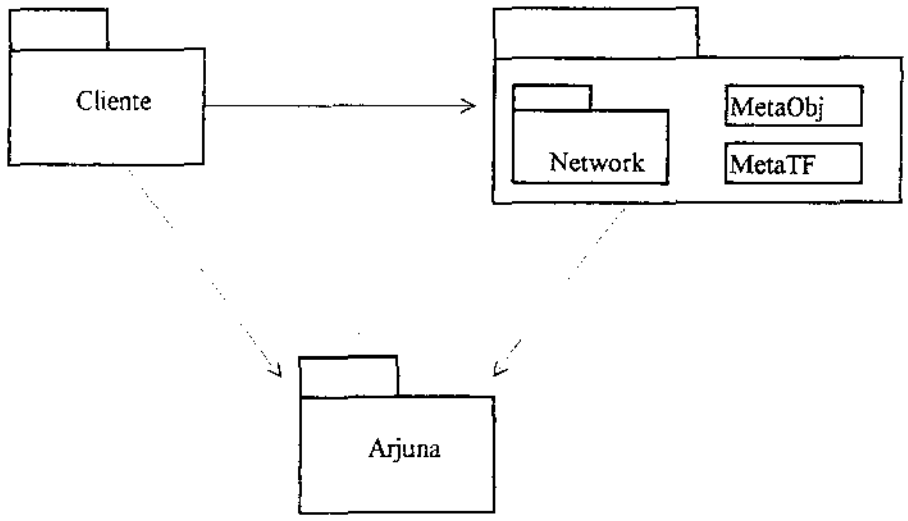



Figura 51: Módulos do Framework FOORD e suas dependências

A Figura 5.2 apresenta uma visão mais detalhada do framework, mostrando sua classes e os relacionamentos entre elas. A seguir descrevemos cada uma das classes do framework. Maiores detalhes de implementação podem ser encontrados no apêndice A .

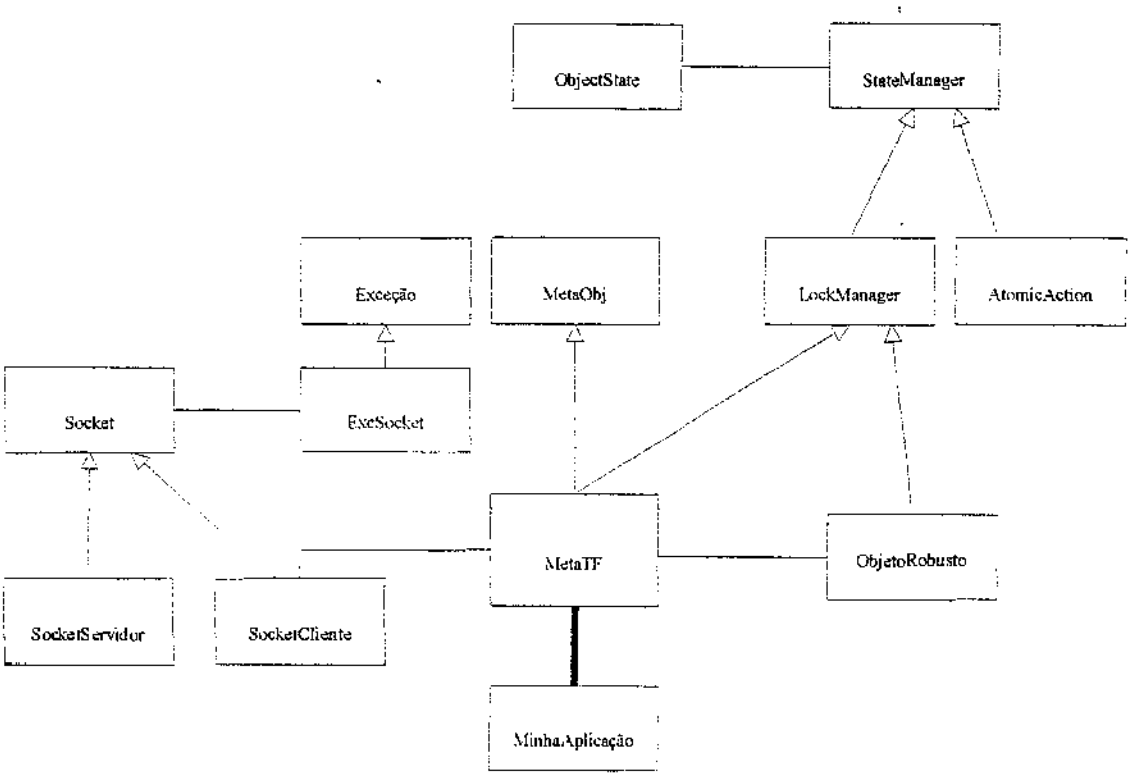



Figura 5.2: Classes do Framework

### 5.2.1 Classe MetaObj

A classe `MetaObj` é a raiz na hierarquia de metaclasses de OpenC++1.2 e toda metaclasse deve ser subclasse de `MetaObj`. A interface da classe `MetaObj`, como mostra a Figura 5.3, consiste num conjunto de operações que estabelecem o protocolo de interceptação de métodos e acesso a variáveis reflexivas. Cada operação, por sua vez, é implementada por um método template e um método hook. Por exemplo, a operação para interceptar chamadas é implementada pelo método template `Meta_MethodCall()` e o método hook `Meta_HandleMethodCall()`.

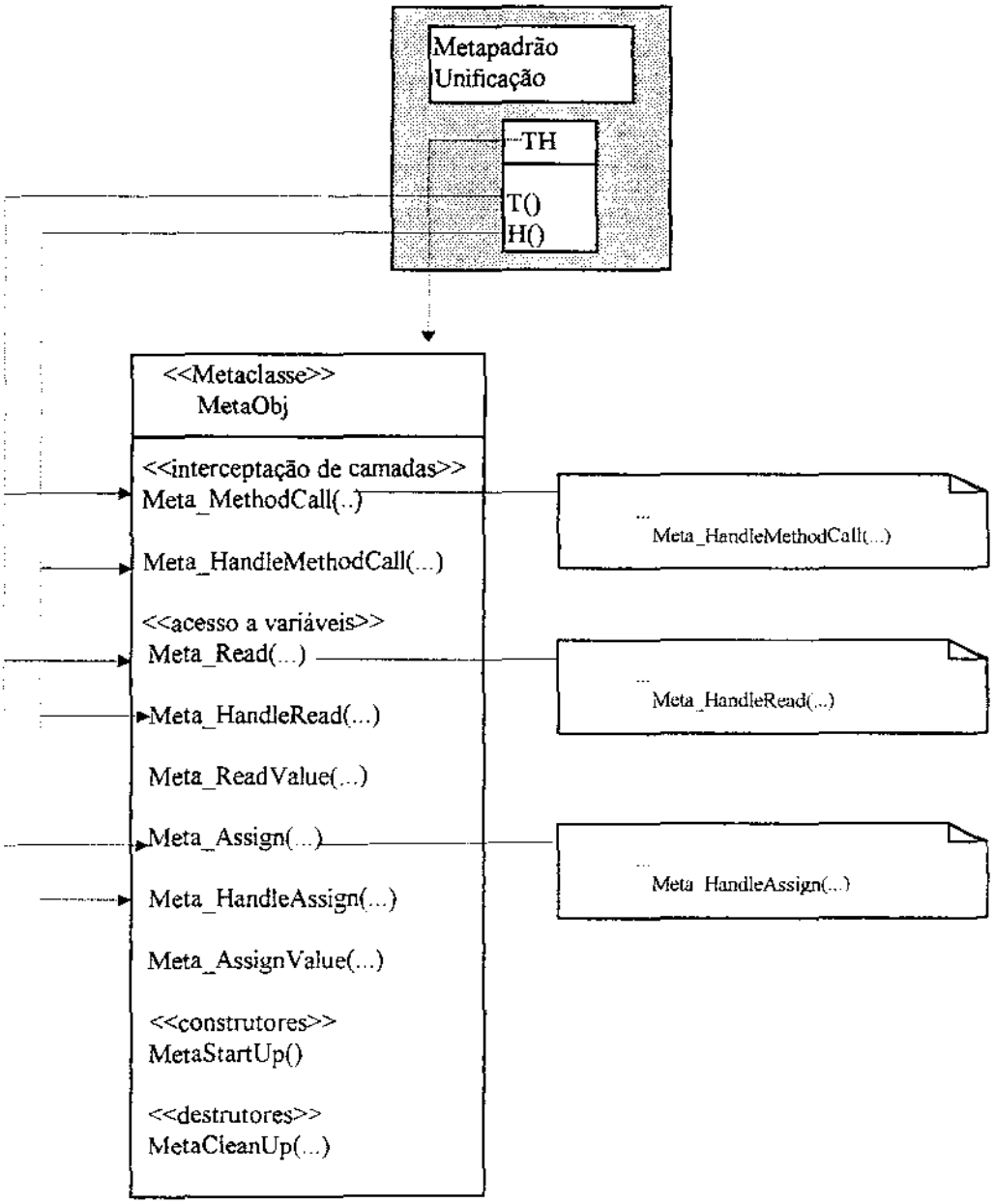



Figura 5.3 Classe `MetaObj`

### 5.2.2 Classes do Arjuna

A classe `StateManager`, suas subclasses (`LockManager` e `AtomicAction`) e a classe `ObjectState`, Figura 5.4, pertencem ao ambiente Arjuna. A classe `AtomicAction` é usada para criar instâncias que fazem uso de ações atômicas. As operações presentes nesta classe (`Begin()`, `Abort()`, `End()`) são usadas para iniciar e manipular ações atômicas.

A classe `StateManager` fornece facilidades para gerenciar objetos persistentes e recuperáveis. Os objetos Arjuna podem ser classificados em três tipos em relação à recuperação de estados: (i) objetos recuperáveis (RECOVERABLE); (ii)recuperáveis e persistentes (ANDPERSISTENT) e não recuperáveis (NEITHER). Objetos NEITHER não podem ser recuperados; objetos RECOVERABLE são recuperáveis mas seu tempo de vida depende da aplicação; objetos (ANDPERSISTENT) são recuperáveis e seu tempo de vida excede o tempo de execução da aplicação. Objetos do framework FOORD são criados automaticamente como ANDPERSISTENT.

A classe `LockManager`, por sua vez, usa as facilidades herdadas da classe `StateManager` e provê controle de concorrência (two-phase locking) para implementar seriabilidade em ações atômicas. A seriabilidade requer que um objeto obtenha um lock de escrita antes que ele seja modificado. Para obter um lock usamos a operação `setlock()`. Os métodos virtuais `save_state()`, `restore_state()` e `type()`, responsáveis pelo mecanismo de restauração de estados, são invocados durante a execução da aplicação e devem ser implementados pelo programador da aplicação, uma vez que a classe `LockManager` não tem acesso, em tempo de execução, à descrição dos objetos C++ na memória e, portanto, não pode implementar uma política default de conversão de objetos.

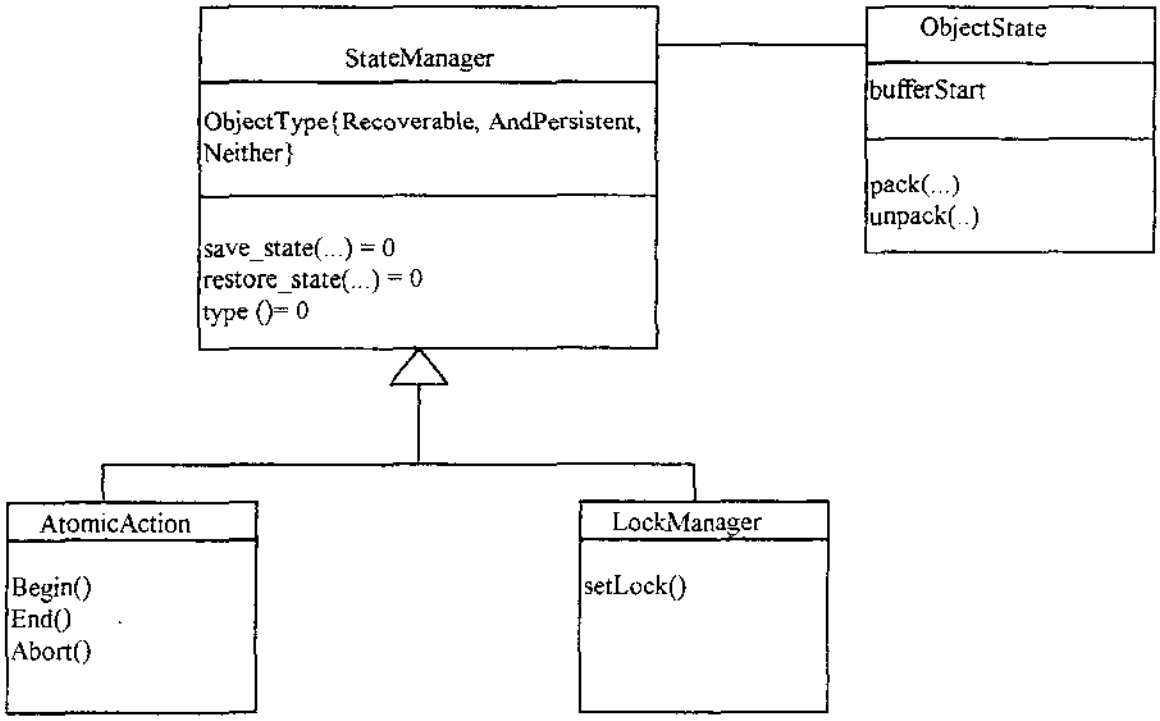



Figura 5.4 Classes do Arjuna

A classe `ObjectState` contém um buffer no qual as instâncias dos tipos primitivos podem ser empacotados/desempacotados continuamente, através da operação `pack()/unpack()`.

### 5.2.3 Classe ObjetoRobusto

`ObjetoRobusto` (Figura 5.5) é uma classe abstrata que define a raiz da hierarquia para objetos tolerantes a falhas da aplicação, portanto, toda classe tolerante a falhas da aplicação deve ser subclasse de `ObjetoRobusto`. O atributo protegido `tecnica_utilizada` guarda a técnica associada com o objeto tolerante a falhas. A interface da classe `ObjetoRobusto` é composta pela operação `retorna_tecnica()`, que retorna a técnica escolhida pelo programador da aplicação, e pelas operações puramente virtuais `seletor()`, `testeaceitacao()`, `marshalling()`, `unmarshalling()`, `save_state()`, `restore_state()` e `type()` cujas implementações devem ser definidas pela aplicação.

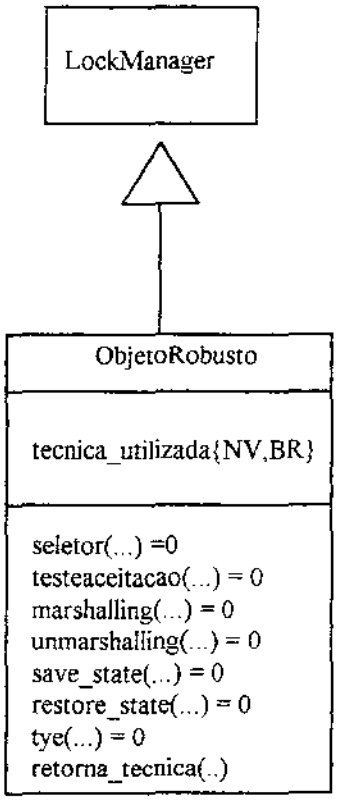



Figura 5.5 Classe `ObjetoRobusto`

A definição da classe `ObjetoRobusto` esta ilustrada no código abaixo:

```
class ObjetoRobusto: public LockManager{
protected:
  enum tecnicaTF tecnica_utilizada;
public:
  ObjetoRobusto(enum tecnicaTF tecnica) : LockManager(ANDPERSISTENT)
   { tecnica_utilizada = tecnica; };
  virtual Boolean testeaceitacao() = 0;
  virtual Boolean seletor(ObjetoRobusto* resp[MAXVER]) = 0;
  virtual char* marshalling() = 0;
  virtual void unmarshalling(char* buff) = 0;
  virtual Boolean save_state(ObjectState& os, ObjectType ot) = 0 ;
  virtual Boolean restore_state(ObjectState& os, ObjectType ot)  = 0;
  virtual  const TypeName type() const = 0;
  enum tecnicaTF retorna_tecnica() { return tecnica_utilizada;};
}
```

### 5.2.4 Classe Excecao

A classe `Excecao` é uma classe abstrata que pode ser especializada pela aplicação pelas próprias classes do framework para representar as exceções geradas durante a execução da aplicação. A classe `ExcSocket` é derivada da classe `Excecao` e representa as exceções de sockets que podem ser geradas durante a execução da aplicação, como por exemplo, exceções de criação, conexão, leitura, escrita, etc, como mostra a Figura 5.6.

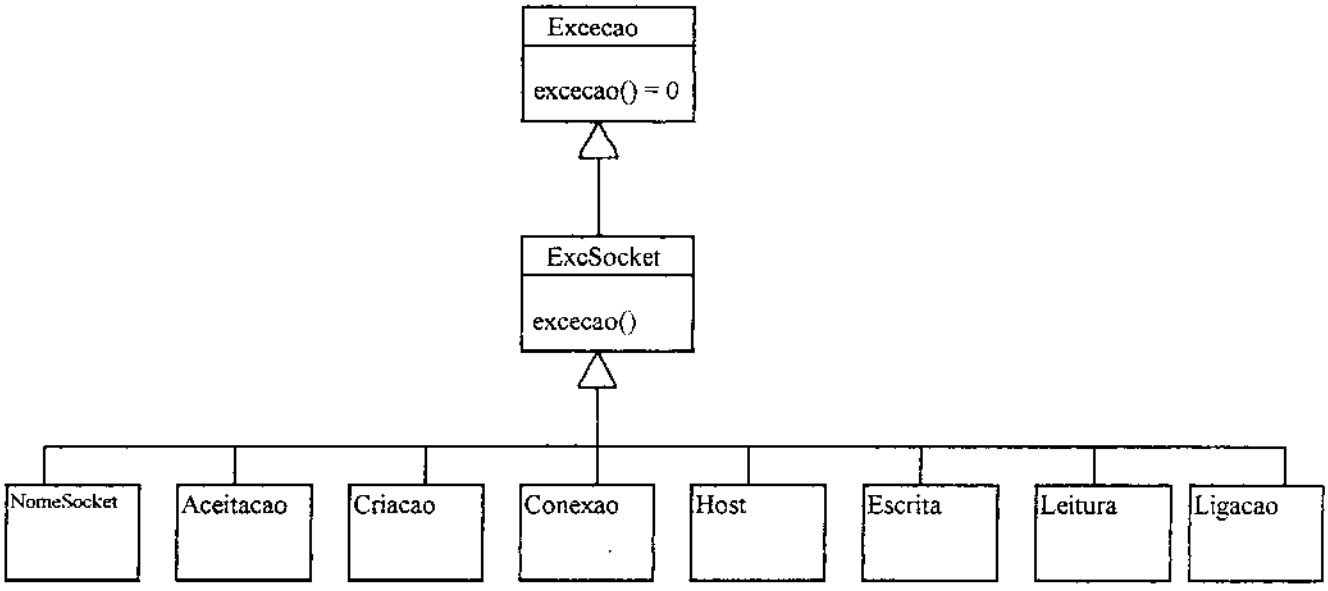



Figura 5.6: Classes `Excecao`

### 5.2.5 Classe Socket

`Socket` é uma superclasse que define os métodos comuns para criação e manipulação de sockets tanto no lado do cliente como no lado do servidor. Estes métodos incluem o método construtor `Socket` e o método Mensagem que implementa as mensagens de erro a serem exibidas como, por exemplo o tratamento de alguma exceção de socket sinalizada. Esta classe é especializada em duas subclasses: `SocketServidor` e `SocketCliente`.

A classe `SocketServidor` é responsável pelos serviços de sockets dos processos servidores (ou versões) que executam em máquinas diferentes. Cada versão recebe uma mensagem contendo o estado do objeto instanciado na aplicação cliente e cria um novo objeto a partir deste estado. A operação requisitada pelo cliente é executada em cima deste novo objeto, podendo eventualmente modificar seu estado. O resultado da aplicação bem como o novo estado do objeto são então retornados ao cliente.

A classe `SocketCliente` é responsável pelos serviços de sockets do processo cliente (aplicação). A operação `envia()` definida nesta classe recebe o objeto da aplicação e invoca o método `marshalling()` para copiar o objeto para um buffer. Após enviar a mensagem ao servidor, o processo cliente é bloqueado até a volta dos resultados obtidos pelo servidor. Em seguida, a operação `recebe()` é invocada e a mensagem vinda do servidor é copiada para um buffer e passada para a aplicação através da invocação do método `unmarshalling()`.

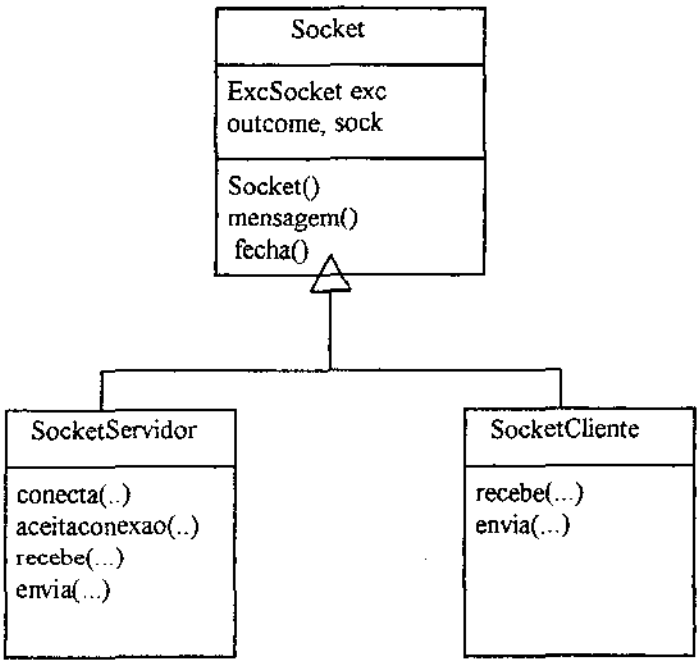



Figura 5.7: Classes `Socket`

### 5.2.6 Classe MetaTF

Esta classe implementa e controla o uso de estratégias de tolerância a falhas para objetos tolerantes a falha reflexivos (Figura 5.8). Os atributos `port` e `maquina` guardam o port associado a cada processo servidor e a maquina onde eles executam. O atributo `obj` é uma referência para o objeto reflexivo da aplicação. Ele é passado como argumento para os métodos da metaclasse, de forma que o objeto do nível base possa ser manipulado pelo meta-nível. O atributo `s` é usado para guardar as conexões entre o cliente e os processos servidores ou versões. O método `Meta_MethodCall()` é herdado de `MetaObj` e é redefinido para interceptar a chamada do método reflexivo e desviar a execução para os métodos que implementam as estratégias de tolerância a falhas(`BlocoRecuperacao()` e `NVersoes()`). Os métodos `save_state()/restore_state()`, invocados dentro do método `BlocoRecuperacao()`, invocam, por sua vez, os métodos `save_state()/restore_state()` definido pelo programador da aplicação de forma a salvar/recuperar o estado do objeto reflexivo. O atributo `contador` guarda o numero de versões usadas pela aplicação e o método `retorna_num_versao()`, invocado pelos método `NVersoes()` e `BlocoRecuperacao()`, retorna o valor deste atributo. A definição da classe `MetaTF` está na próxima Seção.

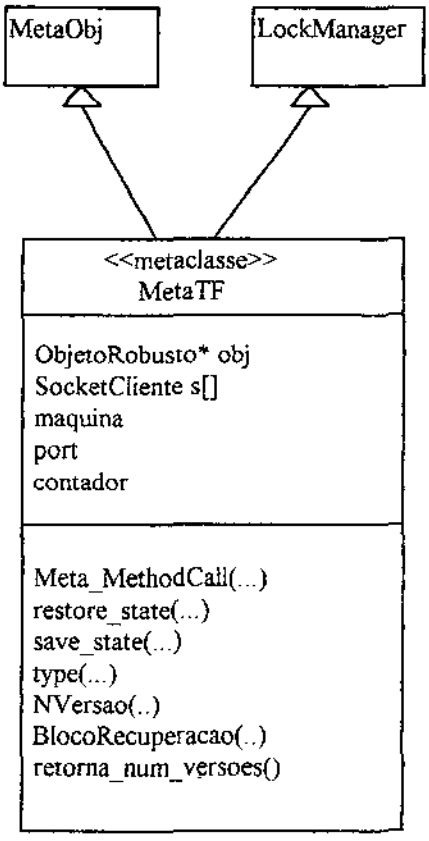



Figura 5.8: Classe `MetaTF`

## 5.3 Aspectos de Implementação da classe MetaTF

MetaTF é derivada de MetaObj e LockManager. Metaobjetos instanciados a partir desta classe devem ser persistentes para permitir restauração de estados. Por isso, no momento de criação de um metaobjeto, o construtor de LockManager é invocado com o argumento ANDPERSISTENT. O construtor MetaTF recebe como parâmetro dois arrays contendo os endereços das máquinas onde as versões executam e seus ports respectivamente. Estes argumentos são fornecidos no momento de criação do objeto reflexivo e são passados automaticamente para o construtor da metaclasse. O código abaixo mostra uma possível definição da classe MetaTF. Uma possível implementação para os métodos save_state() e restore_state() dos objetos da aplicação encontram-se nas Figuras 5.14 e 5.15 respectivamente.

```
class MetaTF: public MetaObj, public LockManager{
private:
  int contador;
  char* maquina[MAXVER];
  int  port[MAXVER];.
  ObjetoRobusto* obj
  virtual Boolean save_state(ObjectState& os, ObjectType ot)
      { obj->save_state(os,ot); }
  virtual Boolean restore_state(ObjectState& os, ObjectType ot)
      { obj->save_state(os,ot); }
  virtual const TypeName type() const
       { return "/StateManager/LockManager/MetaTF" ; }
   int retorna_num_versoes() { return contador;}
public:
    MetaTF(char* mach[MAXVER], int  port[MAXVER]);
    void Meta_MethodCall(Id method_id, Id_category, ArgPac& args, ArgPac&    reply);
    Boolean BlocoRecuperacao(ObjetoRobusto*) ;
    Boolean NVersoes(ObjetoRobusto*) ;
};
```

O método Meta_MethodCall() é herdado da classe MetaObj, e então é redefinido em MetaTF. Este método recebe como argumento uma referencia para o método reflexivo (mid) e seus argumentos (args). Estes argumentos podem ser lidos e tornam-se disponíveis no meta-nível executando-se a operação Pop(), definida para o tipo do argumento. Por exemplo, para o framework FOORD, o argumento do método reflexivo deve ser uma referência para o próprio objeto do nível base de forma que este possa ser manipulado pela método Meta_MethodCall(). Assim usamos o método args.PopPtr() para obtermos o objeto base. Após a execução da operação PopPtr() o atributo obj contem uma referência para o objeto do nível base. Esta referencia é usada para invocar o método retorna_tecnica() que retornará a estratégia de tolerância a falhas associada ao objeto. De acordo com a estratégia escolhida no nível base, os métodos BlocoRecuperacao() ou NVersoes() são executados. Ao final da execução de Meta_MethodoCall(), o resultado é retornado no

argumento `reply`. O trecho de código abaixo refere-se à implementação do método MetaMethodCall().

```
void MetaTF::Meta_MethodCall(Id mid,Id cat, ArgPac& args, ArgPack& reply){
ObjetoRobusto* obj = (ObjetoRobusto*) args.PopPtr();
if (obj->retorna_tecnica() == BR)
    this->BlocoRecuperacao(obj);
else
    this->N-versoes(obj);
args.PushPtr(obj);
Meta_HandleMethodCall(mid,args,reply);}
```

O código para o método `BlocoRecuperacao()` é mostrado abaixo:

```
1  Boolean MetaTF::BlocoRecuperacao(ObjetoRobusto* obj1){
2  SocketCliente* s[MAXVER];
3  obj = obj1;
4  AtomicAction* A;
5  for(int i=0;i<retorna_num_versao();i++){
6     A = new AtomicAction;
7     A->Begin();
8     s[i]->envia(port[i],maquina[i],obj);
9     if(setlock(new Lock(write),0) ==GRANTED)
10        s[i]->recebe(obj)
11    else return (FALSE);
12    if(!obj->testeaceitacao())
13           A->Abort()
14    else{ A->End() ;
15          delete[] s;
16         return TRUE;}
17  }
18  delete[] s;
19  return FALSE ;   // Todas alternativas falharam }
```

O método `BlocoRecuperacao()` recebe um objeto do tipo `ObjetoRobusto` que é enviado para a 1ª. variante através do socket `s[i]` com a chamada do método `envia()` (linha 8). Em Arjuna, a restauração de estados é feita através de uma transação atômica, portanto devemos iniciá-la antes do envio do objeto para cada variante. Uma transação atômica é uma instância da classe `AtomicAction` e quando iniciada, provoca a invocação do método `save_state()`, para copiar o estado do objeto (*this*) antes da transação. Como o objeto que deve ser salvo não é o *this*, mas sim o objeto do nível base, a implementação de `save_state()` em `MetaTF` deve invocar o método `save_state()` do objeto da aplicação, como mostra a o trecho de código da definição da classe `MetaTF`. Quando o resultado de uma variante é retornado

(linha 10), o método `testeaceitacao()` é executado (linha 12). Esse método é definido como puramente virtual na classe `ObjetoRobusto` e deve ser implementado pelo programador da aplicação. Caso o resultado seja aceitável, a transação é validada e o valor TRUE é retornado; caso contrário, a transação é abortada e o estado anterior do objeto é recuperado, através da operação `restore_state()`, e passado para a próxima variante. De maneira similar à operação `save_state()`, para restaurar o estado do objeto da aplicação, o método `restore_state()` em `MetaTF` deve invocar `restore_state()` da aplicação. Se todas as variantes forem executadas e nenhum resultado satisfatório for produzido, o valor FALSE é retornado indicando uma exceção de falha.

A operação `NVersoes()` é implementada de modo similar, mas não é necessário a restauração de estados e, portanto, transações atômicas não são usadas, como mostra o código abaixo.

```
Boolean MetaTF::NVersoes(ObjetoRobusto* obj1){
  SocketCliente* s[MAXVER];
  ObjetoRobusto* resp[MAXVER];
  for(int i=0;i<retorna_num_versoes();i++){
    s[i]->envia(port[i],maquina[i],obj1);
 for(int i=0;i<retorna_num_versoes();i++){
   s[i]->recebe(obj1)
 if(!obj->seletor(resp))
       return FALSE}
 return TRUE;
```

## 5.4 Estudo de Caso: Implementação de uma Pilha Distribuída Tolerante a Falhas

Nesta Seção apresentamos um exemplo de uso do framework apresentado anteriormente. O ambiente no qual foi desenvolvido o exemplo consiste de um conjunto de estações executando sob a plataforma Unix, conectadas através do protocolo TCP/IP. Por ser um framework para aplicações distribuídas, este exemplo usa uma arquitetura cliente/servidor, onde clientes e servidores se comunicam através do uso de sockets. O exemplo implementa uma classe `Pilha` cuja interface pública consiste das operações `empilha()` e `desempilha()` e suas subclasses `PilhaLista` , `PilhaVetor` e `refl_Pilha`. Esta ultima é a classe gerada pelo pre-processador OpenC++, cujas instâncias são objetos reflexivos.

Neste exemplo, o cliente(`refl_Pilha`) utiliza os serviços de tolerância a falhas; os servidores(`PilhaLista` e `PilhaVetor`) implementam os serviços da classe `Pilha`, ou seja, `PilhaLista` e `PilhaVetor` são versões com a mesma interface pública. `PilhaLista` oferece os serviços `empilha()` e `desempilha()` através de uma lista encadeada, enquanto que `PilhaVetor` oferece os mesmos

serviços, porém usando como estrutura de dados um vetor. A comunicação entre cliente e servidores é feita usando-se os serviços das classes `SocketCliente` e `SocketServidor`, como mostra a Figura 5.9.

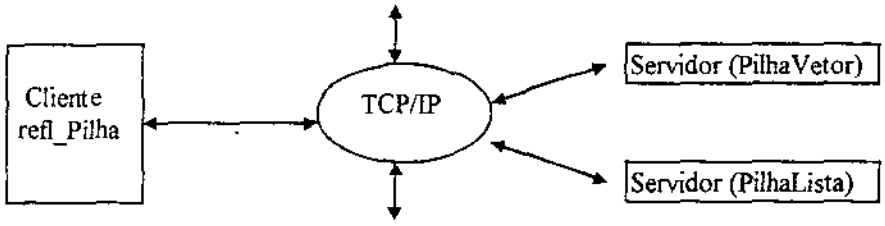



Figura 5.9: Arquitetura Utilizada pelo Sistema

O diagrama de classes da Figura 5.1 foi refeito na Figura 5.10 substituindo-se a Aplicação pela classe `Pilha` e suas versões. A Seção seguinte descreve cada uma dessas classes.

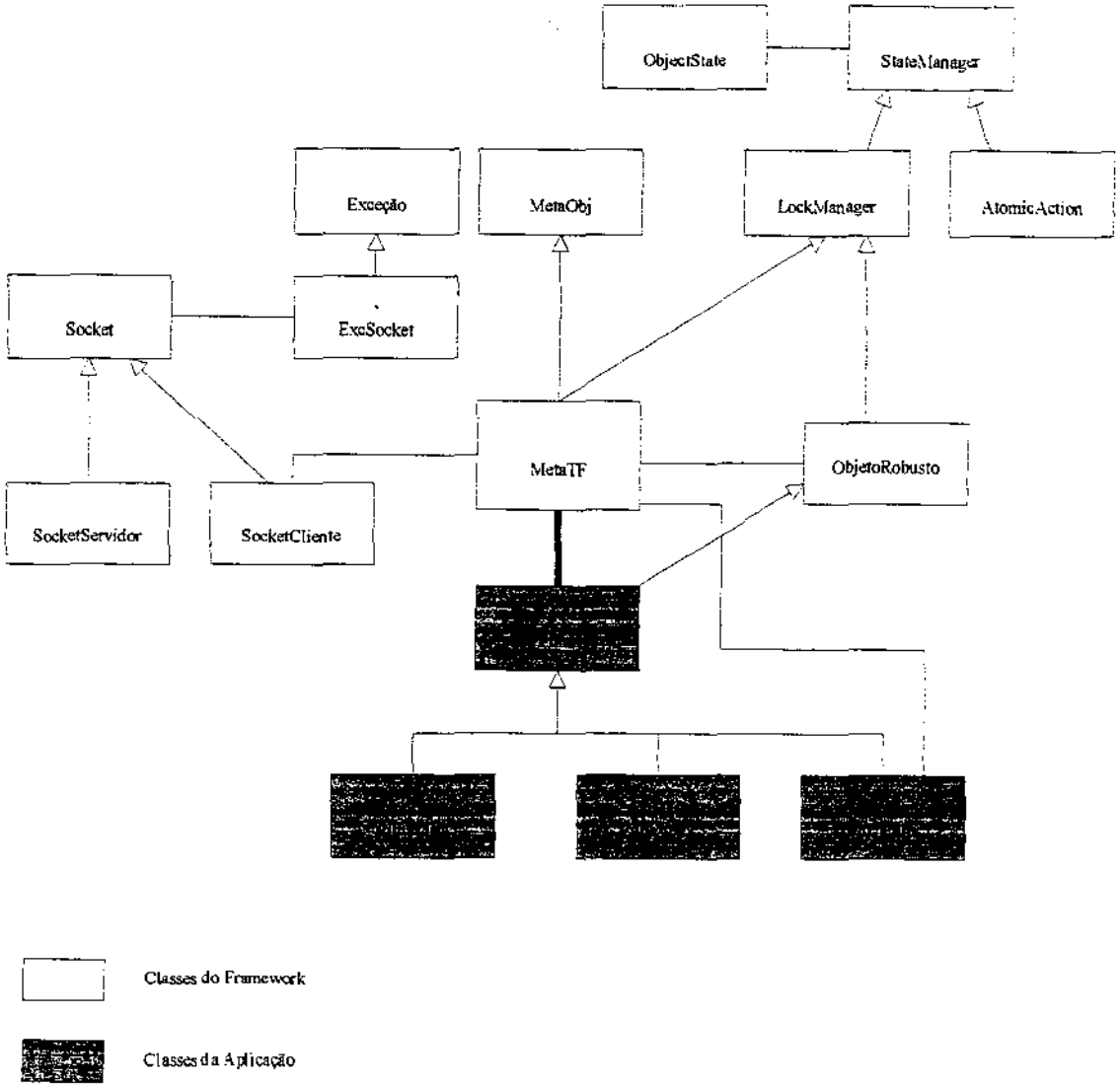



Figura 5.10: Framework para a aplicação `Pilha`

### 5.4.1 Classe Pilha

`Pilha` é a classe da aplicação que utiliza os serviços de tolerância a falhas, portanto ela deve ser subclasse da classe `ObjetoRobusto` e deve também estar associada à metaclasse `MetaTF`, que oferece estes serviços. Cada objeto reflexivo instanciado deve ser associado a uma estrutura de controle(BR ou NV), ou seja, no momento da criação do objeto, o usuário deve indicar qual técnica de tolerância a falhas utilizar para aquele objeto. Isto é feito fornecendo-se a técnica desejada como argumento do construtor da classe `Pilha`. Este construtor por sua vez deve chamar o construtor da classe `ObjetoRobusto` fornecendo-lhe o mesmo argumento. A classe `ObjetoRobusto` se encarrega então de atribuir o argumento ao estado do objeto. A

interface de `Pilha` deve conter também as definições dos métodos `testeaceitacao()`, `seletor()`, `marshalling()`, `unmarshalling()`, `save_state()`, `restore_state()` e `type()`. O método `marshalling()` se encarrega de copiar o estado do objeto (o item a ser colocado na pilha) e a operação desejada (`empilha()`/`desempilha()`) - para a mensagem (buffer) que será enviada ao servidor (Figura 5.11-a). O método `unmarshalling()` é responsável pela operação contrária, ou seja, copiar a mensagem recebida do servidor (contendo o resultado da operação) para o estado do objeto (Figura 5.11-b).

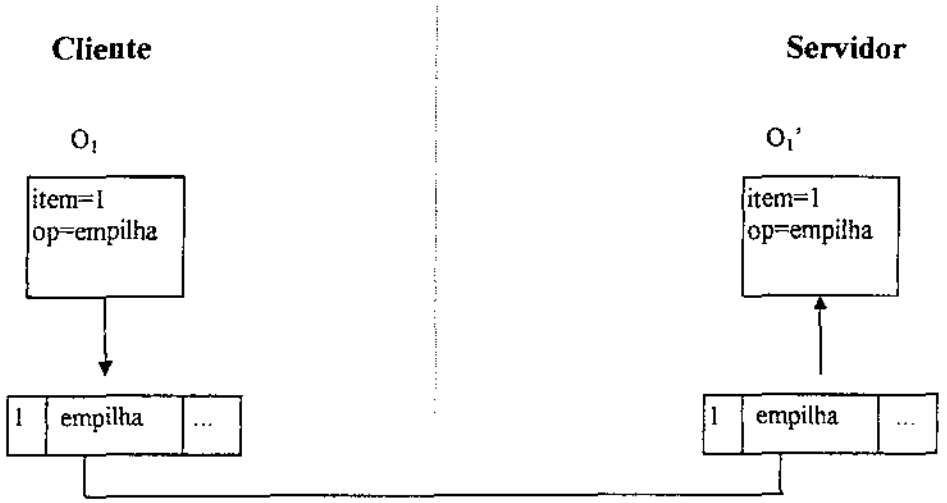



Figura 5.11-a: Transmissão de mensagem do cliente para o servidor

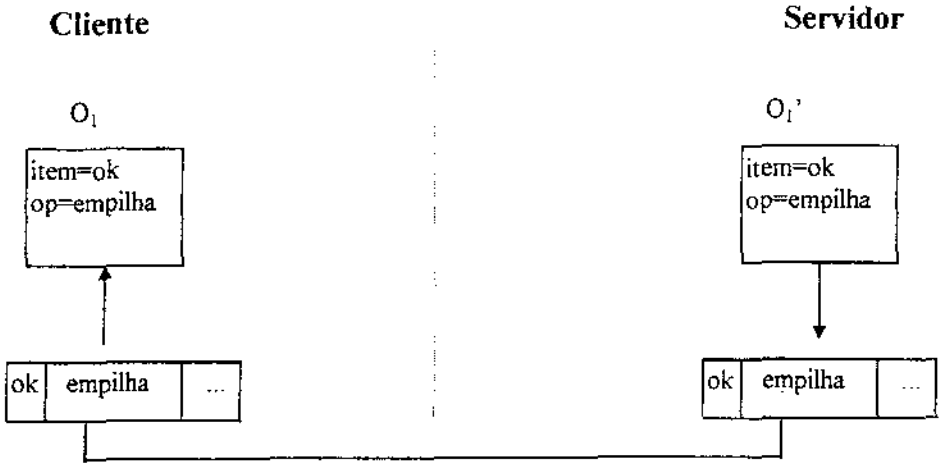



Figura 5.11-b: Transmissão de mensagem do servidor para o cliente

O método `save_state()` e `restore_state()` devem ser definidos para salvar e restaurar o estado do objeto respectivamente. Para salvar o estado do objeto, o método `save_state()` deve conter uma chamada para o método `pack()` (Figura 5.4) para cada atributo que se deseja guardar. Analogamente, o método `restore_state()` deve conter uma chamada para o método `unpack()` (Figura 5.4) para cada atributo salvo pelo método anterior. O método `type()` deve ser definido com a string que representa a hierarquia de herança do objeto.

Uma vez definido estes métodos, definimos os métodos reflexivos através da diretiva **//MOP reflect:** e associamos a classe `Pilha` à metaclasse `MetaTF`. Criamos um objeto reflexivo, lembrando que estes não são instâncias de `Pilha` mas sim de sua subclasse `refl_Pilha`, criada automaticamente por OpenC++. A Figura 5.12 ilustra a hierarquia da classe `Pilha`.

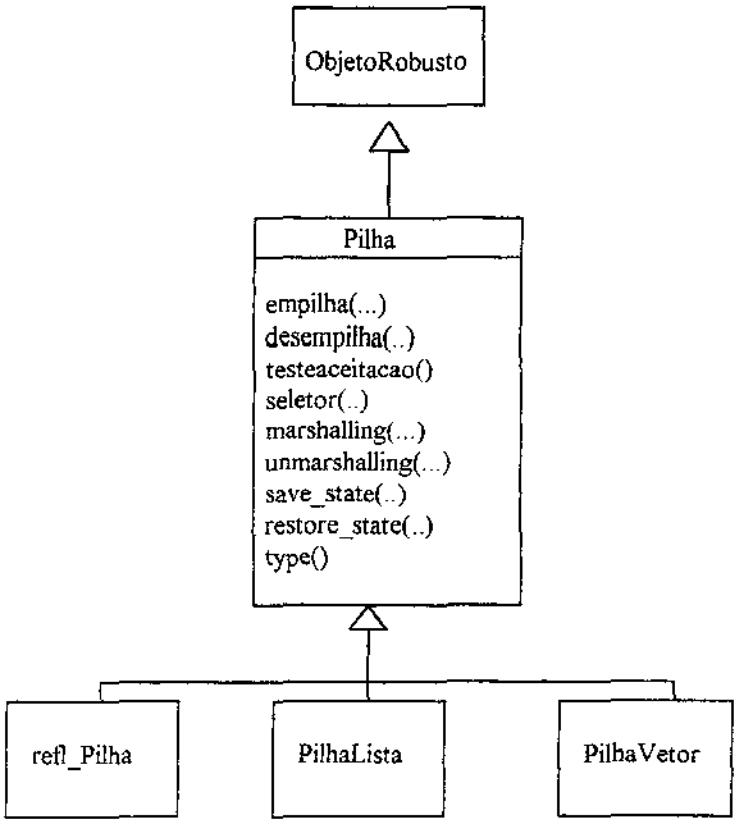



Figura 5.12: Hierarquia da Classe `Pilha`

## 5.5 Pontos Adaptáveis do Framework

O desenvolvimento de um framework requer a identificação dos seus pontos fixos e pontos adaptáveis. No subdomínio de tolerância a falhas, os pontos fixos descrevem as características comuns de diferentes aplicações tolerantes a falhas e os pontos adaptáveis descrevem as partes que podem ser ajustadas ou refinadas nos diferentes contextos de uso do framework proposto . No framework FOORD podemos distinguir os seguintes pontos adaptáveis:

- criação de metaobjetos;
- restauração de estados;
- execução distribuída de versões;
- implementação dos métodos seletores

O formato utilizado nas próximas subseções para descrever os pontos adaptáveis consiste de quatro divisões, baseadas em [48]. Estas divisões descrevem:

(i) o ponto adaptável;
(ii) os problemas relacionados a esta adaptabilidade;
(iii) os requisitos a serem cumpridos pela solução;
(iv) o padrão/metapadrão que cobre a adaptabilidade, bem como sua representação gráfica em notação UML.

Nas notações usadas nas Seções que se seguem, o diamente ao lado das classes faz parte da notação proposta para representar Metapadrões, e portanto não estão representando relacionamentos de agregação, como proposto no Capítulo 2.

### 5.5.1 Ponto Adaptável para Criação de Metaobjetos

Adaptabilidade

Diferentes aplicações possuem diferentes metaobjetos.

Problema

A criação de metaobjetos pode ser dividida em duas fases: a criação do objeto reflexivo e a criação do seu metaobjeto. O objeto reflexivo é criado instanciando-se a classe `refl_Pilha`, gerada por OpenC++ e fornecendo-se os argumento necessários para a criação do objeto bem como de seu metaobjeto. Um dos argumentos do objeto é a estratégia de tolerância a falhas a ele associada, enquanto que os argumentos do metaobjeto são a identificação da máquinas onde as versões executam, bem como os ports dos processos. Quando um objeto do tipo `refl_Pilha` é instanciado, o construtor da classe `Pilha` é invocado. Este por sua vez, invoca o construtor da classe `ObjetoRobusto` fornecendo como parâmetro o tipo de estratégia de tolerância a falhas associada ao objeto (Figura 5.13-a).

Uma vez criado o objeto, inicia-se a criação de seu metaobjeto. O construtor do metaobjeto é invocado automaticamente com seus argumentos. Estes argumentos são guardados nos atributos `maquina` e `port` de `MetaTF` (Figura 5.13-b) a fim de que possam ser manipulados pelas operações `BlocoRecuperação()` e `Nversoes()`.

Padrão/Metapadrão

Os metapadrões Conexão Recursiva 1:1 das Figuras 5.13-a e 5.13-b são aplicados às estruturas para criação de objetos reflexivos e seus metaobjetos respectivamente. Estas figuras ilustram como os componentes destes metapadrões correspondem às classes do framework

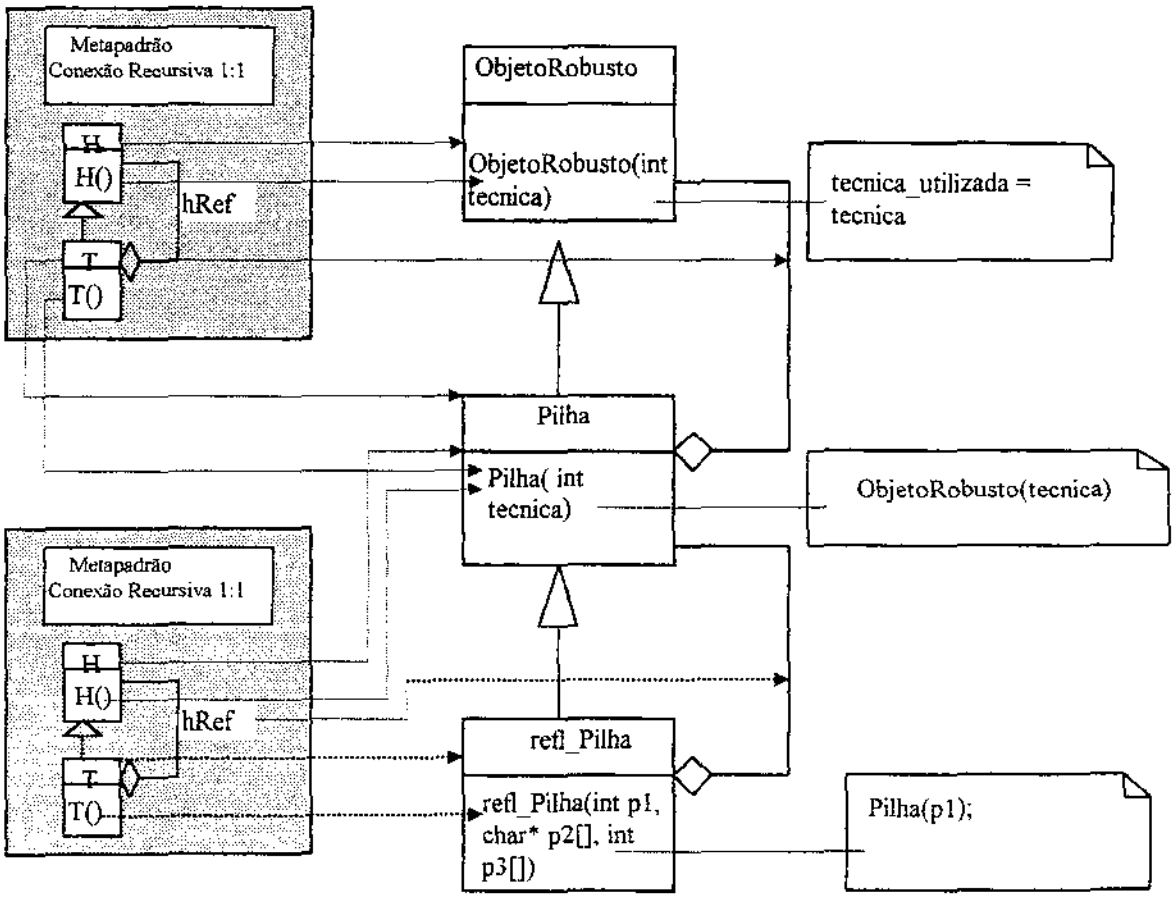



Figura 5.13-a: Metapadrão para a Criação de Objetos Reflexivos

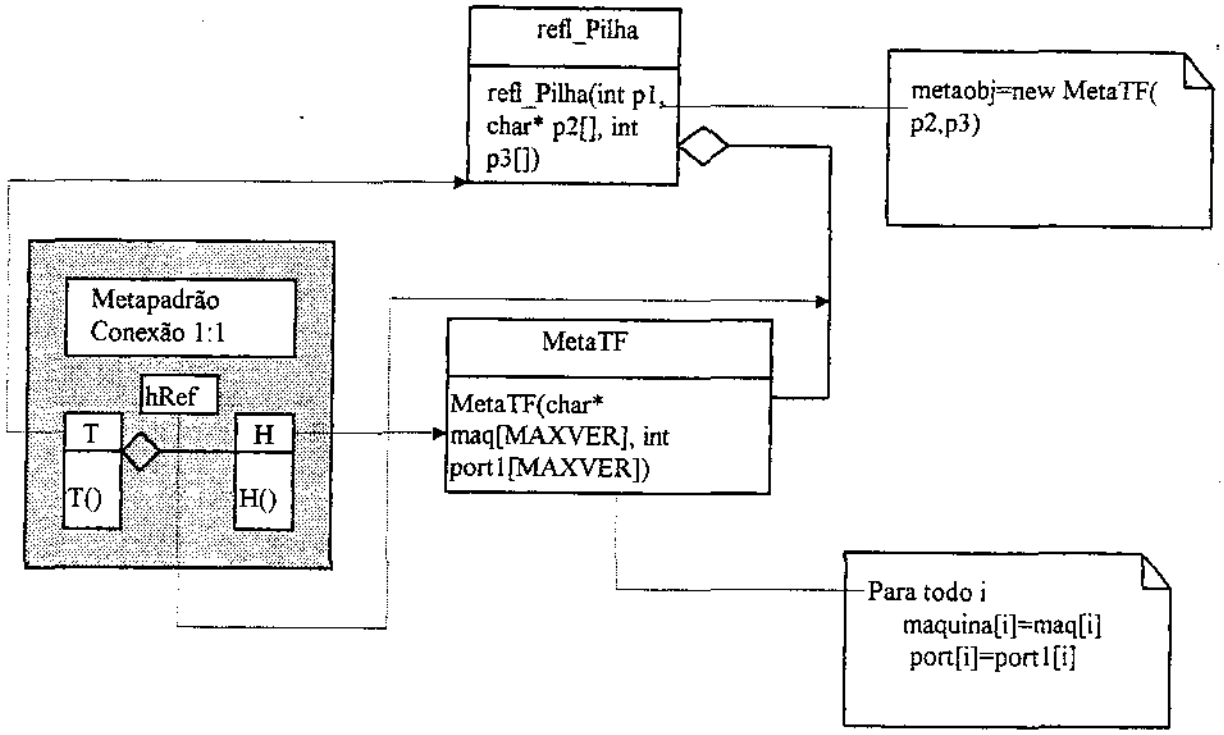



Figura 5.13-b: Metapadrão para a Criação de Metaobjetos

5.5.2 Ponto Adaptável para Restauração de Estados

Adaptabilidade

Diferentes aplicações requerem objetos com estados diferentes.

Problema

O estado dos objetos da aplicação são diferentes e para ser salvo/restaurado, o programador deve definir as operações `save_state()`, `restore_state()` e `type()`. A restauração de estados é feita em duas fases. Na primeira fase, a restauração de estado é iniciada com a instanciação de uma ação atômica que provoca a invocação do método `save_state()` em `MetaTF`. Como o estado a ser salvo é do objeto da aplicação, este método invoca o método `save_state()` da aplicação. O método `save_state()` da aplicação, por sua vez, deve invocar o método `pack()` ( Seção 5.2.2) para cada atributo do objeto (Figura 5.14).

A segunda fase é iniciada quando uma ação atômica aborta provocando a invocação do método `restore_state()` em `MetaTF`. Como ocorre com o método

`save_state()`, este método também invoca o método `restore_state()` da aplicação, que por sua vez, invoca o método `unpack()` (Seção 5.2.2) para cada atributo do objeto (Figura 5.15).

Requisito

Permitir que objetos sejam salvos de maneira flexível e de forma a tornar o framework independente do contexto dos objetos da aplicação.

Padrão/Metapadrão

Os metapadrões Conexão Recursiva 1:1 das Figuras 5.14 e 5.15 são aplicados às estruturas para salvar e restaurar estados de objetos respectivamente. Estas figuras ilustram como os componentes destes metapadrões correspondem às classes do framework

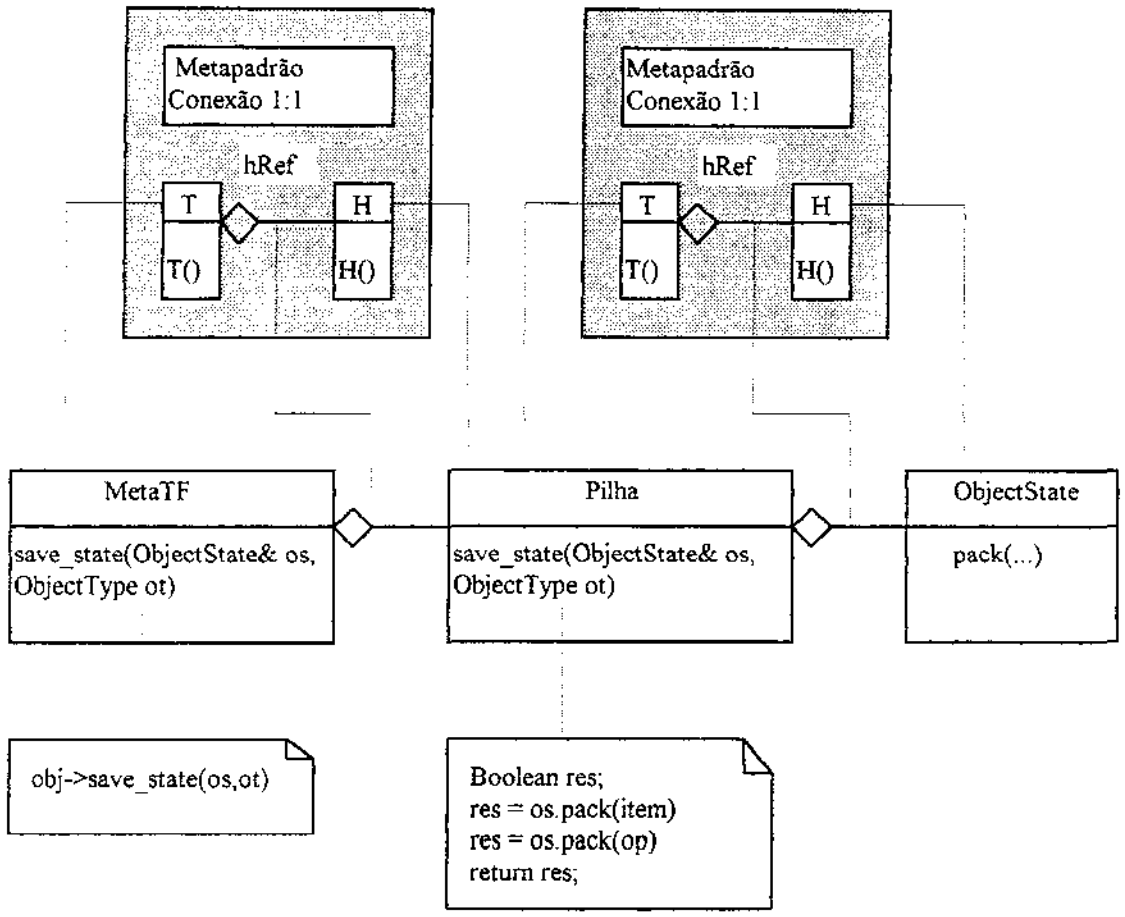



Figura 5.14: Metapadrão para Salvar o Estado de um Objeto

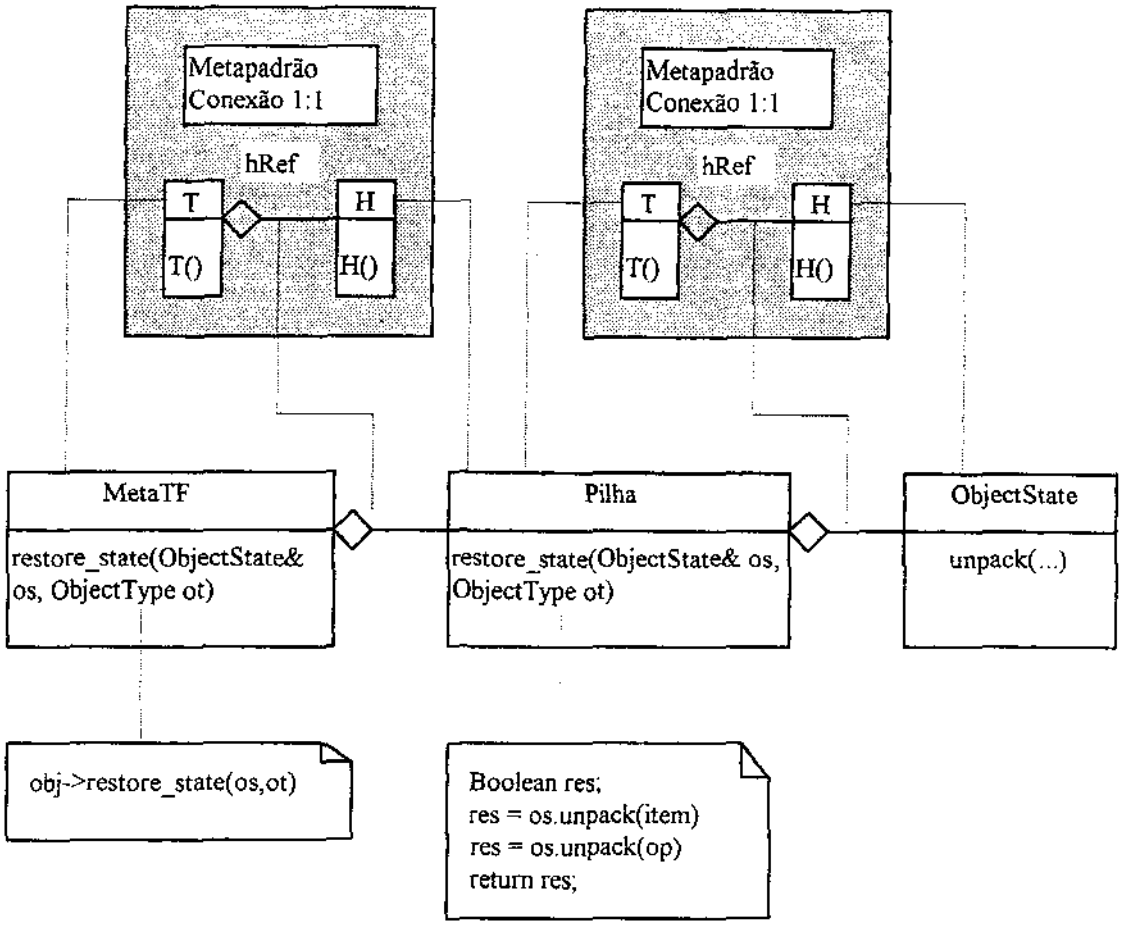



Figura 5.15: Metapadrão para Restaurar o Estado de um Objeto

### 5.5.3 Ponto Adaptável para a Execução Distribuída de Versões

Adaptabilidade

A execução distribuída das versões requer que a aplicação implemente os métodos necessários para a construção da mensagem enviada aos servidores (`marshalling()`), bem como os métodos necessários para transformar as mensagens recebidas pelos mesmos em resultados e estados de objetos (`unmarshalling()`).

Problema

As operações `marshalling()`/`unmarshalling()` aplicadas sobre um objeto robusto têm como objetivo copiar os estados do objeto para um buffer que é então enviado para o processo remoto. Uma vez que a mensagem chega a seu destino, um objeto referente à versão é instanciado com os estados enviados na mensagem. Como mais uma vez estamos trabalhando com estados de objetos precisamos de adaptabilidade.

O estado dos objetos da aplicação são diferentes e para serem linearizados em uma mensagem o programador deve oferecer uma implementação para as operações `marshalling()/unmarshalling()`. O método `marshalling()` é invocado no método `envia()` da classe `SocketCliente` para construir a mensagem que será enviada para as versões. Este método é definido como puramente virtual na classe `ObjetoRobusto` e é herdado e redefinido na classe `Pilha`. A implementação do método `marshalling()` na classe `Pilha`, por sua vez, constrói uma mensagem contendo todos os atributos do objeto (Figura 5.16). O método `unmarshalling()` é invocado no método `recebe()` da classe `SocketCliente` para r transformar a mensagem recebida de um servidor em estados do objeto e no resultado da execução de uma versão. Este método é definido como puramente virtual na classe `ObjetoRobusto` e é herdado e redefinido na classe `Pilha` (Figura 5.17).

Requisitos

Tornar o framework independente do estado dos objetos.

Padrão/Metapadrão

O metapadrão Conexão 1:1 é aplicado à estrutura para criação de objetos reflexivos. As figuras 5.16 e 5.17 ilustram como os Classes deste metapadrão correspondem às classes do framework:

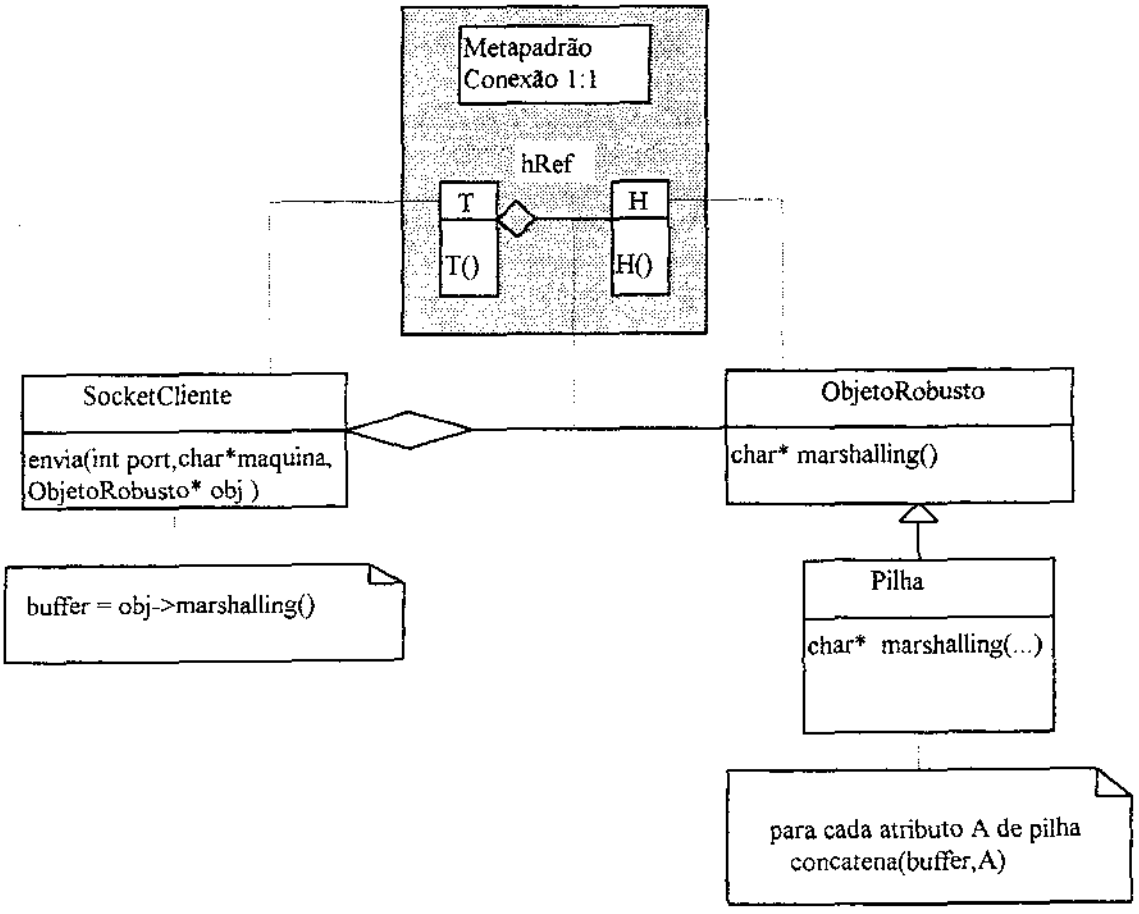



Figura 5.16: Metapadrão para a Operação marshalling

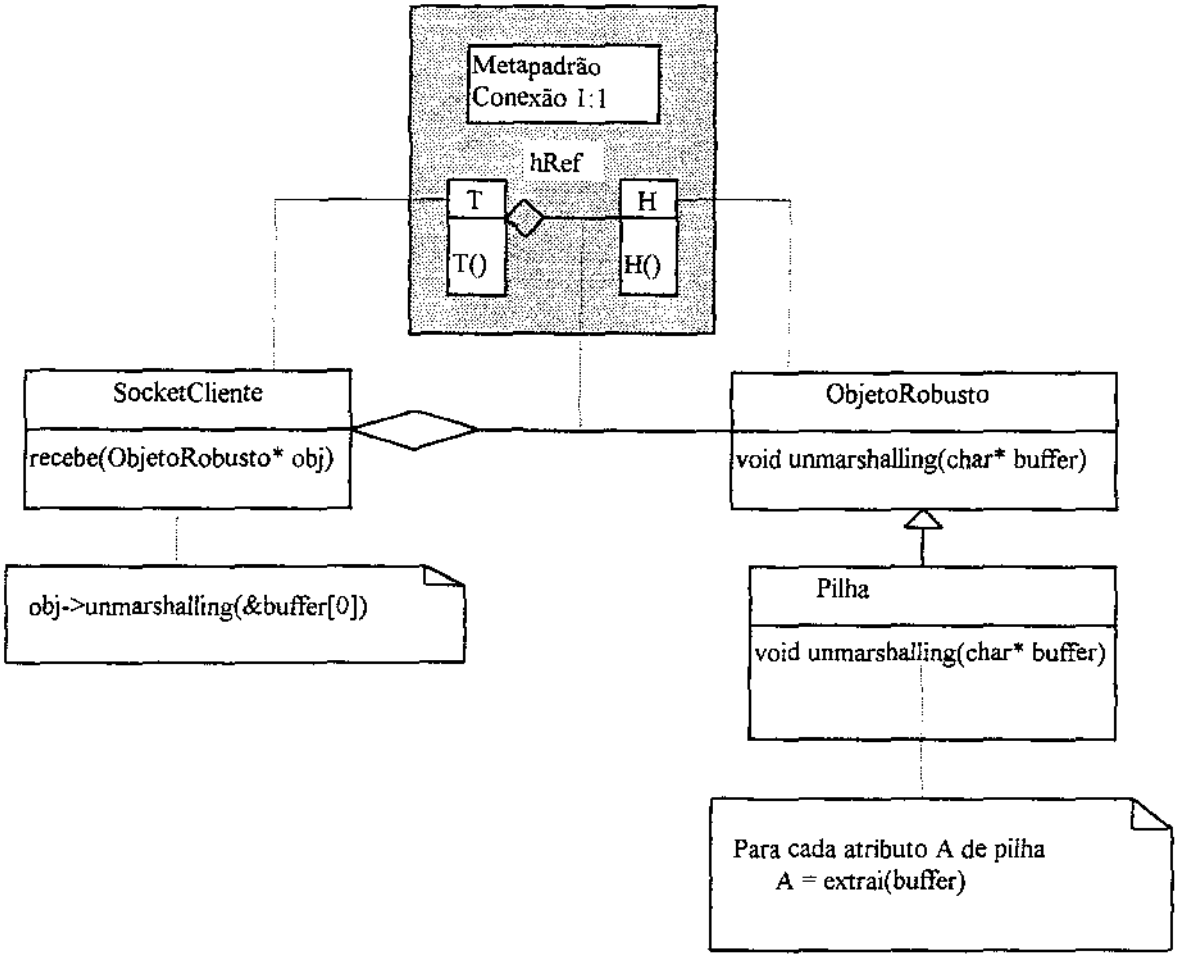



Figura 5.17: Metapadrão para a Operação `unmarshalling()`

### 5.5.4 Ponto Adaptável para a Operação Seletor

Adaptabilidade

Aplicações diferentes têm critérios diferentes para avaliar seus resultados e, portanto, os testes seletores devem ser implementados pelo programador da aplicação.

Problema

As operações `seletor()` e `testeaceitacao()` aplicadas sobre um objeto robusto tem como objetivo determinar se os resultados produzidos por uma ou mais versões são satisfatórios. Uma vez que uma versão retorna o resultado de sua execução para a classe `MetaTF`, a estrutura de controle adequada se encarrega de executar seu método seletor para o resultado obtido. Por exemplo, a operação `testeaceitacao()`

é invocada sobre um objeto do tipo `ObjetoRobusto` no método `BlocoRecuperação()`, o que faz com que o método testeaceitação() definido na classe `Pilha` seja executado. De acordo com o critério de aceitação definido na aplicação, o método retorna TRUE ou FALSE. Entretanto, se o resultado retornado é insatisfatório (FALSE), a ação atômica iniciada ao entrar no bloco de recuperação é abortada (Figura 5.18) e o estado do objeto é restaurado. Analogamente, a Figura 5.19 ilustra os métodos envolvidos na invocação do teste `seletor()` no método `NVersoes()`.

Requisitos

Tornar o framework independente dos testes de aceitação ou seletores da aplicação.

Padrão/Metapadrão

O metapadrão Conexão 1:1 é aplicado à estrutura para avaliação dos resultados produzidos. As figuras 5.18 e 5.19 ilustram como os Classes deste metapadrão correspondem às classes do framework.

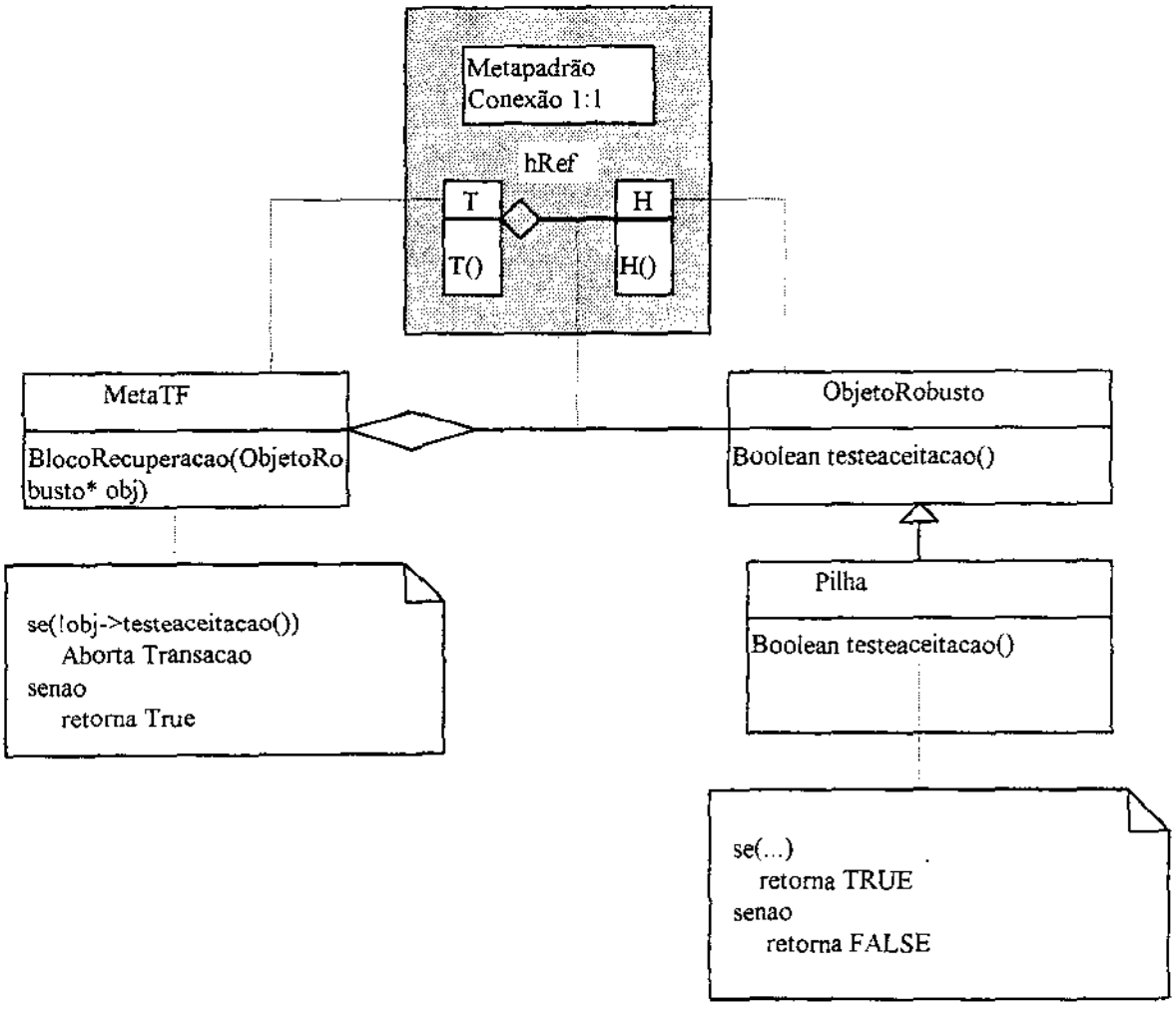



Figura 5.18: Metapadrão para a Operação `testeaceitacao()`

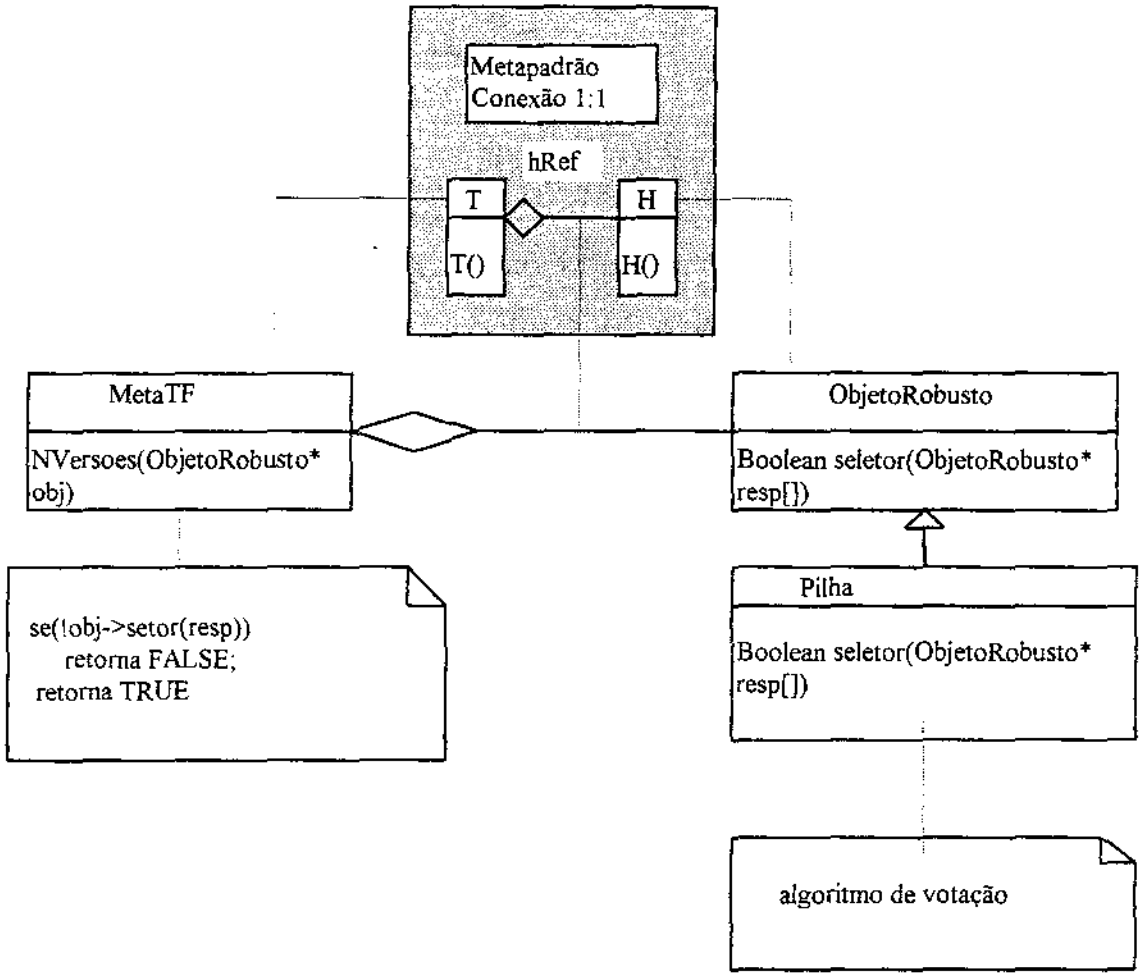



Figura 5.19: Metapadrão para a Operação `seletor()`

## 5.6 Descrição dos Passos para o Uso do Framework

Nesta seção descrevemos os passos necessários para usar o framework FOORD. Os trechos de código abaixo exemplificam os passos.

**Passo 1**: Definir a classe tolerante a falhas (linha 1);
**Passo 2**: Um dos argumentos do construtor da classe tolerante a falhas deve ser um parâmetro `tecnicaTF` (linha 4);
**Passo 3**: O construtor da classe tolerante a falhas deve conter uma chamada para o construtor do `ObjetoRobusto` passando como parâmetro a técnica associada ao objeto (linha 4);
**Passo 4**: Definir os métodos `testeaceitacao()` e `seletor()` (linhas 5 e 6);
**Passo 5**: Definir os métodos `marshalling()` e `unmarshalling()` (linhas 7 e 8);

**Passo 6**: Definir os métodos `save_state()`, `restore_state()` e `type()` (linhas 9, 10 e 11). `save_state()` contém uma chamada `os.pack()` para cada atributo do objeto. `restore_state()` tem uma chamada `os.unpack()` para cada atributo do objeto. `type()` contém uma string que representa a hierarquia de classes do objeto.
**Passo 7**: Definir os métodos reflexivos usando a diretiva **//MOP reflect:** (linha 12). O argumento do método reflexivo deve ser uma referência para o próprio objeto;
**Passo 8:** Associar a classe tolerante a falhas à metaclasse `MetaTF` (linha 15);
**Passo 9:** Implementar as versões como processo que fazem uso dos serviços de sockets disponíveis em `SocketServidor`. Definir operações `marshalling()` e unmarshalling para os servidores. O Framework permite um número máximo de 10 versões.
**Passo 10:** Executar as versões em máquinas remotas;
**Passo 11:** Criar o objeto reflexivo a partir da classe `refl_<nome da classe tolerante a falhas>`. Fornecer os argumentos normais para a construção do objeto juntamente com os ports e a identificação das máquinas onde executam as versões (linha 18);
**Passo 12:** Executar a aplicação cliente.

```
1 class Pilha: public ObjetoRobusto{
2 int item;
3 public:
4 Pilha(enum tecnicaTF tecnica) { ObjetoRobusto(tecnica);};
5 Boolean testeaceitacao();
6 Boolean seletor(ObjetoRobusto* resp[MAXVER]);
7 char* marshalling();
8 void unmarshalling(char* buff);
9 Boolean save_state(ObjectState& os, ObjectType ot) { Boolean res=os.pack(item);};
10 Boolean restore_state(ObjectState& os, ObjectType ot) {Boolean res=os.unpack(item);};
11 const TypeName type() const {return "StateManager/LockManager/ObjetoRobusto/Pilha/refl_Pilha";
12 //MOP reflect:
13 virtual void empilha(int item, ObjetoRobusto* or)
14 virtual void desempilha(ObjetoRobusto* or) }

15 //MOP reflect class Pilha : MetaTF;

16 main(){
...
17 //obter port e maquina onde executam as funcoes
...
18 ref_Pilha  pilha(BR,maquina,port);
..
}
```

A diretiva //MOP reflect: define a operação `empilha()` e `desempilha()` como métodos reflexivos e a diretiva //MOP reflect class associa a classe `Pilha` à sua metaclasse `MetaTF` que implementa os métodos BR e NV. A classe `MetaTF` redefine o método Meta_MethodCall da classe `MetaObj`. Este método recebe como argumento o objeto que recebeu a invocação do método reflexivo.

Quando o método reflexivo é invocado, a chamada é interceptada pela classe `MetaTF`. Esta invoca o método `BlocoRecuperacao()` ou `NVersoes()` de acordo com a estrutura de controle associada ao objeto reflexivo no momento de sua criação. A classe `MetaTF` se encarrega de criar o socket e estabelecer a conexão entre os processos cliente e servidor bem como acionar a técnica de tolerância a falhas escolhida pelo usuário. Isto está esquematizado na figura 5.20.

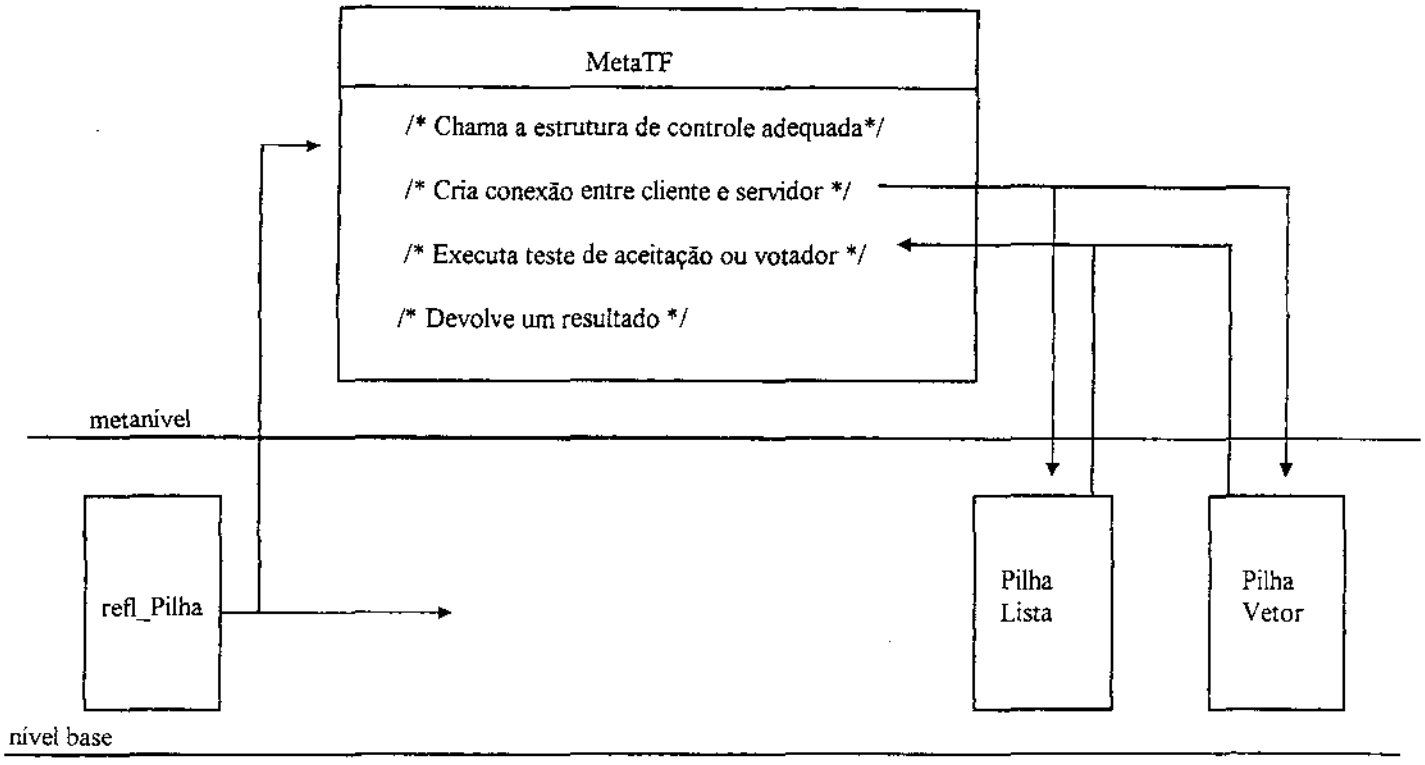



Figura 5.20: Interceptação do Método Reflexivo e Ativação de Versões

## 5.7 Avaliação do Framework Proposto

Nesta Seção o framework proposto foi comparado com um framework similar para uma arquitetura orientada a objetos não reflexiva[38]. A Figura 5.21 ilustra o framework orientado a objetos utilizado no nosso experimento. Neste framework, os serviços de tolerância a falhas são fornecidos pela classe `TF`, enquanto que objetos tolerantes a falha são instâncias da classe `PilhaRobusta`, que encapsula as chamadas para as estratégias de tolerância a falhas(BR,NV). As demais classes são as mesmas do framework reflexivo.

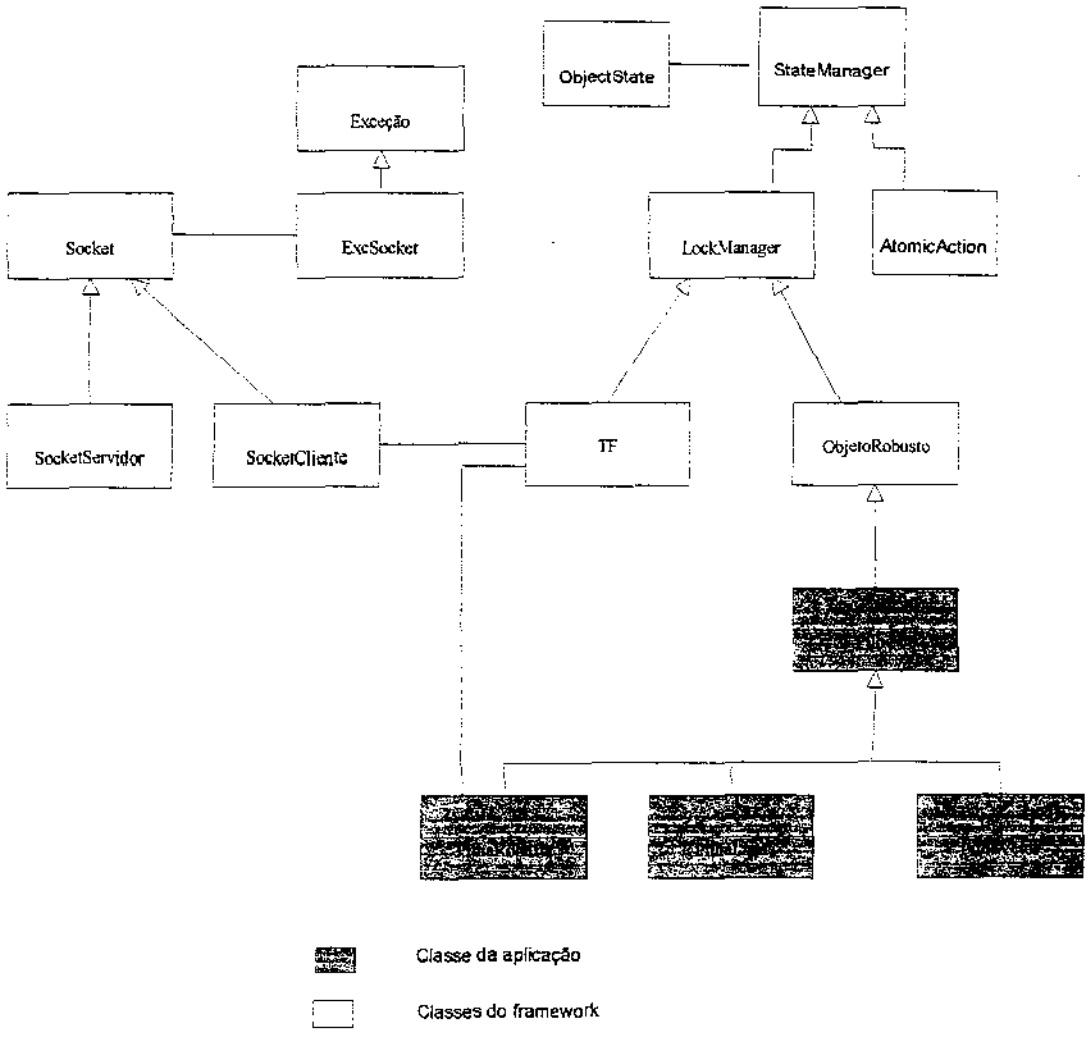



Figura 5.21: Um Framework Orientado a Objetos Distribuído para Tolerância a Falhas

Os tempos de execução para cada uma das abordagens foram coletados com o objetivo de avaliar o desempenho do mecanismo de reflexão computacional. O tempo foi coletado em segundos , e como o exemplo foi implementado para um ambiente distribuído, o tempo gasto com a comunicação de rede não foi desprezado. Os tempos gastos para o armazenamento de 200 elementos numa pilha robusta foram coletados levando-se em consideração os seguintes casos:

(1) N-versões,
(2) Bloco de Recuperação onde a 1ª variante é executada com sucesso e;
(3) Bloco de Recuperação onde 1ª variante é executada com fracasso e a 2ª é executada com sucesso.

Para cada abordagem, esses três casos foram considerados e os tempos obtidos estão ilustrados na tabela 5.1. Nesta tabela notamos que o tempo de execução do mecanismo de N-versões é relativamente menor quando comparado com o de blocos de recuperação, pois além das variantes serem executadas concorrentemente, a implementação de N-versões não utiliza as transações atômicas do Arjuna. Já no bloco de recuperação, o ambiente Arjuna grava o objeto em disco para validar a transação. Isso implica num tempo de execução bem maior do que no caso de N-versões. Entretanto, se compararmos os tempos gastos nas duas colunas, podemos concluir que o custo adicional introduzido pela reflexão computacional é baixo. Os tempos de execução demonstraram que o overhead extra proveniente da materialização de mensagens dos mecanismos de reflexão é baixo dependendo das atividades executadas no meta-nível, e podem até ser considerado desprezível quando comparado a tempos gastos na utilização de outros recursos de sistema, como comunicação da rede e acesso a memória estável.

| Casos | Framework OO | Framework OO Reflexivo |
|---|---|---|
| N-versões | 1.9s | 2.1s |
| BR 1ª variante sucesso | 7.9s | 8.7s |
| BR 2ª variante sucesso | 10.2s | 13.6s |

Tabela 5.1: Tabela dos Tempos de Execução para N-versões e BR

## 5.8 Resumo

A arquitetura reflexiva do framework FOORD para a implementação de aplicações tolerantes a falhas é organizada em níveis visando a reutilização e a transparência de serviços. Naturalmente, a transparência total de serviços não foi atingida em parte devido ao próprio domínio da aplicação, bem como devido à certas exigências de algumas técnicas de tolerância a falhas (como o fato do usuário fornecer as versões e os algoritmos de seleção). O próprio pre-processador OpenC++1.2 apresentou certas deficiências que comprometeram a transparência desejada.

No framework FOORD, cada objeto do nível base pode ser associado estaticamente a um metaobjeto que controla apenas acesso a atributos e chamadas de métodos. Metaobjetos são instâncias da classe `MetaTF` que fornece os serviços de tolerância a falhas. A raiz de metaclasse é a classe `MetaObj` que manipula objetos do tipo `ObjetoRobusto`, raiz para classes tolerantes a falhas da aplicação. As classes `SocketCliente` e `SocketServidor` oferecem os serviços de comunicação para clientes e servidores respectivamente.

Neste capítulo apresentamos um pequeno estudo de caso usando o framework FOORD. Um exemplo concreto de uma aplicação tolerante a falhas foi implementada usando os serviços do framework e os tempos de execução foram coletados e comparados com os tempos de execução do mesmo implemento em uma arquitetura não reflexiva. Os tempos de execução demonstraram que o overhead dos mecanismos de reflexão é baixo quando comparado a tempos gastos na comunicação da rede e acesso a memória estável.

Esperamos assim, termos contribuído para mostrar que reflexão computacional é uma técnica viável e útil para o desenvolvimento de aplicações tolerantes a falhas.

# Capítulo 6
# Conclusões e Trabalhos Futuros

## 6.1 Desenvolvimento da pesquisa

Este trabalho faz parte de um estudo mais abrangente para a definição de uma arquitetura de software reflexiva genérica que atualmente está sendo desenvolvida pelo Laboratório de Sistemas Distribuídos (LSD) do Instituto de Computação da Unicamp. Em particular, o desenvolvimento desta pesquisa centralizou-se na definição de uma arquitetura reflexiva para o desenvolvimento de aplicações tolerantes a falhas de software, visando principalmente simplificar o desenvolvimento destas aplicações e conseqüentemente torná-las mais confiáveis. A pesquisa iniciou-se com o estudo de técnicas de programação orientada a objetos, apresentadas no capítulo 2, seguindo para o estudo de técnicas clássicas de tolerância a falhas de software e a importância do modelo de objetos para desenvolver tais técnicas. Isto foi apresentado no capítulo 3. Este capítulo traz as conclusões e trabalhos futuros dessa dissertação de mestrado, bem como as suas contribuições. A Seção 6.2 traz alguns trabalhos relacionados à tolerância a falhas; a Seção 6.3 mostra algumas considerações sobre técnicas de tolerância a falhas; a Seção 6.4 mostra algumas considerações sobre o modelo de orientação a objetos; as Seções 6.5 e 6.6 discutem sobre reflexão computacional e arquitetura reflexiva para tolerância a falhas respectivamente.

## 6.2 Trabalhos Relacionados

Esta seção discute alguns trabalhos encontrados na literatura relacionados com o desenvolvimento de software tolerante a falhas. A primeira subseção traz dois softwares não orientados a objetos. As demais seções estão relacionadas com técnicas de orientação a objetos e reflexão computacional.

### 6.2.1 Software Não Orientado a Objeto

Na literatura podemos citar dois sistemas experimentais não orientados a objetos, implementados para tolerar falhas de software: Polylith[41], um ambiente que dá suporte para a execução de programas usando bloco de recuperação e n-versões para softwares não orientados a objeto; e RMP[2], um ambiente que promove a separação clara entre aplicação dos mecanismos de tolerância a falhas ainda que esta separação seja feita de maneira "ad hoc" e pouco extensível.

### 6.2.2 Software Orientado a Objeto

Poucos trabalhos na literatura têm explorado técnicas orientadas a objeto para a provisão de tolerância a falhas de software. Rubira e Stroud[44] mostram como implementar recuperação de erros com avanço e retrocesso em C++ dentro do contexto de programação seqüencial. O trabalho de Rubira[46] propõe dois esquemas genéricos orientados a objeto para implementar componentes ideais com diversidade de projeto. Xu et al[59] integra dois conceitos complementares -transações e conversações - para implementar tolerância a falhas de software.

Dentre os trabalhos que exploram reflexão computacional e orientação a objetos para tolerância a falhas de software destacam-se Xu et al[58], Zorzo[65] e MOFT[27]. Em Xu et al[58] encontramos uma proposta de arquitetura reflexiva para tolerância a falhas, mas resultados concretos não são apresentados. Zorzo et al[65] apresenta uma avaliação de desempenho de uma arquitetura reflexiva usando um mecanismo de injeção de falhas. Por outro lado, a proposta do MOTF é desenvolver aplicações pertencentes ao domínio de tolerância a falhas, sob a ótica do modelo de orientação a objeto, implementados em uma arquitetura reflexiva. O objetivo principal deste trabalho ao incorporar os princípios de reflexão computacional ao modelo de orientação a objetos é salientar 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.

O nosso trabalho propôs a soma de alguns aspectos dos trabalhos acima citados, sintetizando-os em um framework que promove a integração de uma arquitetura orientada a objetos (Arjuna) com uma arquitetura reflexiva de forma a garantir a provisão de tolerância a falhas de software, contribuindo assim para a sedimentação de uma arquitetura reflexiva para este domínio. Como o trabalho de Zorzo et al[65], o nosso trabalho também se propõe a avaliar o desempenho e os tempos envolvidos em uma aplicação reflexiva, de forma a questionar a viabilidade do seu uso. Mas ao contrário do trabalho de Zorzo, nossa arquitetura se torna mais complexa ao incorporar o ambiente Arjuna ao framework, adicionando outros elementos como ações atômicas, persistência e restauração de estados às atividades executadas no meta-nível.

## 6.3 Considerações sobre Técnicas de Tolerância a Falhas

Embora a demanda por software tolerantes a falha seja crescente em áreas de aplicações críticas, sua aplicabilidades ainda é limitada em outras áreas, possivelmente devido ao seu alto custo de desenvolvimento. Acreditamos que uma maneira de se reduzir este custo é através da reutilização adequada de componentes que provêm estratégias de tolerância a falhas de propósito geral, sem ao mesmo tempo, comprometer a diversidade de projetos, que é essencial para tolerar falhas durante a fase operacional do software.

No capítulo 5, mostramos que o framework FOORD possibilita a reutilização de componentes através de duas formas: definição clara e transparente das interfaces entre objetos tolerantes a falhas e objetos não tolerantes a falha, e a possibilidade de reflexão sobre objetos individuais da aplicação e o controle de interações com outros objetos sem, entretanto, interferir em seu encapsulamento.

## 6.4 Considerações sobre o Modelo de Objetos

A adequação do modelo de orientação a objetos para desenvolver softwares tolerantes a falhas já foi explicitada em alguns trabalhos, como em [42][44][46]. De maneira geral, podemos citar o mecanismos de herança e encapsulamento que permitem reutilização de código e facilidades para a implementação de diversidade de projeto através da concretização de classes abstratas através de implementações distintas, separando os aspectos comportamentais e estruturais. Mas, além dessas soluções clássicas, o modelo de orientação a objetos admite novas soluções. O modelo de orientação a objetos adapta-se melhor ao conceito de reflexão computacional do que qualquer outro paradigma de programação. O modelo de classe e metaclasse fornece um ótimo gancho para reflexão computacional e a habilidade das linguagens orientadas a objeto de se ajustarem e modificarem facilmente encoraja seu uso. Também o uso de padrões de projeto foi uma consideração importante à medida que estes facilitaram a descrição e acomodação de tarefas do framework.

## 6.5 Considerações sobre Reflexão Computacional

A idéia de termos linguagens orientadas a objetos que permitem que programas reflitam sobre sua computação tem merecido a atenção de vários pesquisadores. Este interesse por reflexão concretizou-se quando Pattie Maes apresentou sua linguagem reflexiva 3KRS. Desde então, vários outros sistemas e linguagens com capacidades reflexivas foram criados, com o objetivo de fornecer um fórum aceitável sobre algumas questões envolvendo reflexão computacional. Em particular, neste projeto de pesquisa procuramos identificar as seguintes questões:

- O que é reflexão computacional?

- Qual a relação entre nível base e meta-nível?
- Existe uma relação forte entre reflexão e programação orientada a objeto?
- A nível de implementação, como linguagens compiladas superam o "gap" entre compilação estática e sistemas reflexivos?
- Em quais sistemas seria vantajoso o uso de reflexão?

As respostas que encontramos para tais questões foram:

- Reflexão computacional pode ser definida como a habilidade de um sistema de representar, operar e tratar sua própria computação da mesma forma que ele representa, opera e trata sua aplicação.
- Um sistema pode estar em três situações diferentes em relação à conexão entre o nível base e o meta-nível. Em um extremo da relação não existe nenhuma conexão entre os dois níveis. Esta é a situação onde se encontram os sistemas tradicionais. Em outro extremo da relação, existe uma ligação completa entre nível base e meta-nível, como é o caso de 3-Lisp. Entre os dois extremos, existe uma conexão parcial, onde reside a maior parte dos problemas do mundo real e onde acreditamos deva ser o centro de atenção de nossas pesquisas.
- Programação orientada a objetos é a mais indicada para incorporar reflexão e implementar aplicações reflexivas, como vimos na seção anterior.
- Em relação a reflexão em linguagens compiladas, algumas pesquisas[10,11,12] mostram que é possível uma reflexão limitada nestas linguagens. Uma consideração cuidadosa do que é permitido refletir na linguagem pode melhorar a eficiência do sistema.
- Como existe um overhead associado aos mecanismos de reflexão, acreditamos que estes mecanismos são praticáveis desde que o domínio selecionado seja tal que o overhead introduzido pela técnica de reflexão seja desprezível em relação ao overhead associado com a funcionalidade implementada no meta-nível[11].

## 6.6 Considerações sobre Arquitetura Reflexiva e Trabalhos Futuros

Vimos no decorrer desta dissertação que reflexão é uma técnica vantajosa para sistemas que necessitam de requisitos administrativos para implementar e controlar os requisitos funcionais da aplicação. Sistemas tolerantes a falha podem se beneficiar com uma arquitetura reflexiva, uma vez que os requisitos relacionados à tolerância a falhas estão mais associados à máquina virtual que implementa a aplicação do que a própria aplicação. No capítulo 5 apresentamos alguns benefícios da arquitetura reflexiva para sistemas tolerantes a falhas e apresentamos o framework FOORD para desenvolver aplicações tolerantes a falhas de software reflexivas e distribuídas. Ainda no Capítulo 5, o framework foi usado como solução para um problema de programação, com o objetivo de se avaliar a eficiência e viabilidade de se usar tal arquitetura para sistemas tolerantes a falhas. O resultado obtido mostrou que para o domínio de tolerância a falhas, quase sempre relacionado com restauração de estados de objetos, distribuição, persistência, ações atômicas e controle de concorrência, o uso de reflexão computacional é viável,

uma vez que o overhead para manutenção destas funcionalidades é bem maior que o overhead de reflexão. Além disso, esse overhead pode ser reduzido se a aplicação for cuidadosamente projetada, uma vez que o programador tem a possibilidade de escolher quais objetos da aplicação serão reflexivos. Esperamos assim termos contribuído para a série de pesquisas que estão sendo feitas sobre reflexão computacional.

Como sugestões de trabalhos futuros, seria interessante a avaliação de uma arquitetura reflexiva para tolerância a falhas em um PMO mais flexível que o fornecido em OpenC++1.2. Dentro desta perspectiva, o PMO presente em OpenC++2.0 é bastante interessante.

# Apêndice

A .1 Componente MetaObj
CLASS MetaObj
DESCRIPTION A classe MetaObj é a raiz na hierarquia de metaclasses de OpenC++. Toda metaclasse é uma subclasse de MetaObj. Os métodos disponíveis nesta classe utilizam os seguintes argumentos:

---

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;

---

PROVIDED_INTERFACE

OPERATIONS [Métodos publicos definidos pela classe]

**void Meta_MethodCall(Id method_id, Id_category, ArgPac& args, ArgPac& reply)**

DESCRIPTION Este método é redefinido pela metaclasse para estender o comportamento do método reflexivo. Os três primeiros argumentos referem-se ao identificador, categoria e argumentos de chamada do método reflexivo. O argumento reply especifica o valor retornado pelo método chamado e que será retornado ao cliente.

**void Meta_Read(Id var_id, Id category, ArgPac& reply)**

DESCRIPTION Este método é executado para ler o valor de uma variável reflexiva. var_id e category referem-se ao identificador e categoria da variável. reply contém uma referência à variável lida.

**void Meta_StartUp()**

DESCRIPTION Este método é usado para definir construtores especiais a nível de meta-objetos e é invocado logo após a instanciação de um objeto reflexivo e seu meta-objeto correspondente.

**void Meta_CleanUp()**

DESCRIPTION Este método é usado para definiu destrutores especiais a nível de meta-objetos e é invocado logo após a destruição de um objeto reflexivo e seu meta-objeto correspondente.

OPERATIONS [Métodos protegidos definidos pela classe]

**void Meta_HandleMethodCall(Id method_id, ArgPac& args, ArgPac& reply)**

DESCRIPTION Este método executa o método de nível base identificado por method_id. Os argumentos de entrada do método reflexivo são armazenados em args e o resultado em reply. Este método é chamado dentro do método Meta_MethodCall definido pelo programador.

**void Meta_HandleAssign(Id var_id, ArgPac& args)**

DESCRIPTION Este método atribui uma referencia para o valor de args à variável identificada em var_id. Este método normalmente é chamado pelo método Meta_Assign definido pelo programador.

**void Meta_HandleRead(Id var_id, ArgPac& reply)**

DESCRIPTION Este método devolve através de reply uma referência ao valor armazenado na variável identificada em var_id. Este método é executado pelo método Meta_Read.

**void Meta_AssignValue(Id var_id, ArgPac& args)**

DESCRIPTION Este método atribui o valor (e não a referência para o valor) armazenado em args à variável reflexiva var_id.

**void Meta_ReadValue(Id var_id, ArgPac& reply)**

DESCRIPTION Este método copia o valor(e não a referência para o valor) da variável identificada em var_id para o argumento reply.

**const char* Meta_GetClassName()**

DESCRIPTION Este método retorna uma referência para o nome da classe reflexiva.

**const char* Meta_GetVarName(Id var_id)**

DESCRIPTION Este método retorna uma referência para o nome da variável reflexiva.

**const char* Meta_GetMethodName(Id method_id)**

DESCRIPTION Este método retorna uma referência para o nome do método reflexiva.

SUPERCLASSES Não possui superclasses

SUBCLASSES MetaTF

END_CLASS MetaObj

A .2 Classe LockManager

CLASS LockManager

DESCRIPTION A classe LockManager pertence ao ambiente Arjuna e provê ações atômicas e controle de concorrência para implementar restauração de estados. Para fazer uso de ações atômicas, objetos da classe AtomicAction são instanciados. Estes objetos fornecem operações Begin, End e Abort para iniciar e manipular ações atômicas. Objetos controlados por ações atômicas devem ser derivados da classe LockManager. A propriedade de seriabilidade requer que um objeto obtenha um lock de escrita antes que ele seja modificado. Para isto, usamos a operação setlock. Os métodos save_state, restore_state e type são invocados durante a execução da aplicação e devem ser implementados pelo programador da aplicação, uma vez que LockManager não tem acesso em tempo de execução à descrição do objeto C++ na memória e portanto não pode implementar uma política default de conversão de objetos. Os métodos disponíveis nesta classe utilizam os seguintes argumentos:

---

LockResult {GRANTED,REFUSED,RELEASED}
ObjectType{RECOVERABLE,ANDPERSISTENT,NEITHER}
ObjectState& os objeto onde são empacotados tipos padrões através da operação pack e unpack

---

PROVIDED_INTERFACE

OPERATIONS [Métodos públicos definidos pela classe]

**virtual Boolean save_state(ObjectState& os, ObjectType ot)**

DESCRIPTION Este método salva o estado de um objeto para que possa ser recuperado posteriormente

**virtual Boolean restore_state(ObjectState& os, ObjectType ot)**

DESCRIPTION Este método restaura o estado de um objeto previamente salvo pelo método anterior.

**virtual const TypeName type() const**

DESCRIPTION Este método armazena a hierarquia de herança como uma string para uma classe determinada.

**LockResult setlock( Lock *toSet, int, unsigned int)**

DESCRIPTION Este método permite que um objeto obtenha um lock de escrita antes que ele seja modificado.

SUPERCLASSES StateManager

SUBCLASSES MetaTF

END_CLASS LockManager

A .3 Classe ObjetoRobusto

CLASS ObjetoRobusto

DESCRIPTION ObjetoRobusto é uma classe abstrata que define a raiz da hierarquia para objetos tolerantes a falhas da aplicação. A variável protegida TecnicaUtilizada guarda a técnica associada com o objeto tolerante a falhas.

---

tecnicaTF tecnica_utilizada {NV,BR}

---

PROVIDED_INTERFACE

OPERATIONS [Métodos públicos definidos pela classe]

**virtual Boolean seletor(ObjetoRobusto* resp[])**

DESCRIPTION Este método deve ser definido pela aplicação para efetuar o algoritmo de decisão usado no n-versões

**virtual Boolean testeaceitacao()**

DESCRIPTION Este método deve ser definido pela aplicação para efetuar o teste de aceitação usado no bloco de recuperação.

**virtual char* marshalling()**

DESCRIPTION Este método deve ser definido pela aplicação para implementar o empacotamento dos estados do objeto para serem enviado às versões que executam em outras máquinas.

**virtual void unmarshalling(char*)**

DESCRIPTION Este método deve ser definido pela aplicação para implementar o desempacotamento dos estados do objeto e o resultado da execução da versão.

**virtual Boolean save_state(ObjectState& os, ObjectType ot)**

DESCRIPTION Este método salva o estado de um objeto para que possa ser recuperado posteriormente

**virtual Boolean restore_state(ObjectState& os, ObjectType ot)**

DESCRIPTION Este método restaura o estado de um objeto previamente salvo pelo método anterior.

**virtual const TypeName type() const**

DESCRIPTION Este método armazena a hierarquia de herança como uma string para uma classe determinada.

**virtual int retorna_tecnica()**

DESCRIPTION Este método retorna a técnica{NV,BR} associada ao objeto tolerante a falha.

SUPERCLASSES LockManager

SUBCLASSES classes da aplicação que implementam serviços críticos

END_CLASS ObjetoRobusto

A .4 Classe Excecao
CLASS Excecao
DESCRIPTION A classe Exceções é uma classe abstrata que pode ser especializada pela aplicação pelas próprias classes do framework para representar as exceções geradas durante a execução da aplicação.
PROVIDED_INTERFACE
OPERATIONS [Métodos públicos definidos pela classe]
**virtual int exceções()**
DESCRIPTION Este método retorna o tipo de uma exceção que foi sinalizada.
SUPERCLASSES Não possui superclasse
SUBCLASSES ExcSocket e exceções definidas na aplicação.
END_CLASS Excecao

A .5 Classe Socket
CLASS Socket
DESCRIPTION Socket é uma superclasse que define os métodos comuns para criação e manipulação de sockets tanto no lado do cliente como no lado do servidor.
PROVIDED_INTERFACE
OPERATIONS [Métodos públicos definidos pela classe]
**Socket()**
DESCRIPTION Este método construtor cria um socket tanto no lado cliente como servidor
**void mensagem(const char* )**
DESCRIPTION Este método implementa as mensagens de erros a serem exibidas como tratamento de alguma exceção de socket sinalizada.
**void close()**
DESCRIPTION Fecha o socket
SUPERCLASSES Não possui superclasse
SUBCLASSES SocketServidor e SocketCliente
END_CLASS Exceções

A .6 Classe ExcSocket
CLASS ExcSocket
DESCRIPTION Esta classe representa as exceções de sockets que podem ser geradas durante a execução da aplicação
PROVIDED_INTERFACE
OPERATIONS [Métodos públicos definidos pela classe]
**virtual int exceção()**
DESCRIPTION Este retorna o tipo de exceção de socket gerada.
SUPERCLASSES Exceções
SUBCLASSES Criacao, Conexao, Host, Escrita, Leitura, Ligacao, NomeSock, Aceitacao.
END_CLASS ExcSocket

A .7 Classe SocketServidor

CLASS SocketServidor

DESCRIPTION Esta classe é responsável pelos serviços de sockets dos processos servidores (ou versões) que executam em máquinas diferentes. Cada versão recebe uma mensagem contendo o estado do objeto instanciado na aplicação cliente e cria um novo objeto a partir deste estado. A operação requisitada pelo cliente é executada em cima deste novo objeto, podendo eventualmente modificar seu estado. O resultado da aplicação bem como o novo estado do objeto devem ser retornados ao cliente.

PROVIDED_INTERFACE

OPERATIONS [Métodos públicos definidos pela classe]

**void conecta()**

DESCRIPTION Este método

**void aceitaConec()**

DESCRIPTION

**int recebe(char[])**

DESCRIPTION Recebe a mensagem enviada pelo cliente

**void envia()**

DESCRIPTION Envia a mensagem ao cliente

SUPERCLASSES Socket

SUBCLASSES Não tem subclasses

END_CLASS SocketServidor

A .8 Classe SocketCliente

CLASS SocketCliente

DESCRIPTION Esta classe é responsável pelos serviços de sockets do processo cliente (aplicação). Cada versão recebe uma mensagem contendo o estado do objeto instanciado na aplicação cliente e cria um novo objeto a partir deste estado. A operação requisitada pelo cliente é executada em cima deste novo objeto, podendo eventualmente modificar seu estado. O resultado da aplicação bem como o novo estado do objeto devem ser retornados ao cliente.

PROVIDED_INTERFACE

OPERATIONS [Métodos públicos definidos pela classe]

**void recebe(ObjetoRobusto*)**

DESCRIPTION Recebe a mensagem enviada pelo servidor

**void envia(int port,char* machine[],ObjetoRobusto*)**

DESCRIPTION Envia a mensagem ao servidor

SUPERCLASSES Socket

SUBCLASSES Não tem subclasses

END_CLASS SocketCliente

A .9 Classe MetaTF

CLASS MetaTF

DESCRIPTION Esta classe (metaclasse) implementa e controla o uso de estratégias de tolerância a falhas para objetos tolerantes a falha reflexivos.

PROVIDED_INTERFACE

OPERATIONS [Métodos públicos definidos pela classe]

**void Meta_MethodCall(Id method_id, Id_category, ArgPac& args, ArgPac& reply)**

DESCRIPTION Este método é executado no momento em que o método reflexivo é invocado e é responsável por ativar a estratégia de controle adequada associada ao objeto reflexivo.

**Boolean BlocoRecuperacao(ObjetoRobusto*)**

DESCRIPTION Este método implementa a técnica de bloco de recuperação

**Boolean NVersoes(ObjetoRobusto*)**

DESCRIPTION Este método implementa a técnica de n-versões

**virtual Boolean save_state(ObjectState& os, ObjectType ot)**

DESCRIPTION Este método invoca o método save_state definido na classe do método reflexivo.

**virtual Boolean restore_state(ObjectState& os, ObjectType ot)**

DESCRIPTION Este método invoca o método restore_state definido na classe do método reflexivo.

**virtual const TypeName type() const**

DESCRIPTION Este método define a string de hierarquia de herança até a classe MetaTF.

SUPERCLASSES MetaObj e LockManager

SUBCLASSES Não tem subclasses

END_CLASS MetaTF

# Referências


[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Massachussetts, 1986.

[2] M.Ancona et al. *Reflective Arquitetures for Reusable Fault-Tolerant Software*. XXI Latin American Conf. on Informatics, Canela, RS, Brasil, Agosto 1995.

[3] M.Ancona et al. *A System Architeture for Fault Tolerance in Concurrent Software*, IEEE Computer (23)10:23-32, Outubro 1990.

[4] T.Anderson. Fault Tolerant Computing. Em Resilient Computing Systems. T.Anderson (Ed.), Chapter 1. Collins Professional and Techincal Books, 1985.

[5] A . Avizienis. *The N-Version Approach to Fault Tolerant Software*. IEEE Transaction on Software Engineering, SE-11(12): 1491-1501, Dezembro 1985.

[6] K.J.Birman. *Replication and Fault Tolerance in the Isis Systems*. ACM Operating Systems Reviews, vol 19, 1985, pp 79-86.

[7] G.S. Blair. *Basic Concepts III (Types, Abstract Data Type and Polymorphism)*. Em Object-Oriented Languages, Systems and Apllications, G.S. Blair, J. Gallagher, D. Hutchison & D. Shepherd (Eds.), Chapter 4, Pitman, 1991.

[8] G.Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings Publishing Company, Inc., 1991.

[9] G. Booch, J. Rumbaugh e J. Jacobson. *Unified Modelling Language*. Versão 1.0, Rational Software Corporation, Janeiro 1997.

[10] L.Cardelli & P.Wegner. *On Understanding Types, Data Abstraction and Polymorphism*. Computing Surveys, 17(4): 471-522, Dezembro 1985.

[11] S.Chiba & T.Masuda. *Designing an Extensible Distributed Language with a Meta-Level Architecture*. Em Proccedings of the 7th European Conference on Object-Oriented Programming, ECOOP'93, Kaiserslautern, Germany Lecture Notes in Computer Science, 707: 482-501, Oscar M. Nierstrasz (Ed.), Julho 1993, Springer-Verlag

[12] S.Chiba. *OpenC++ Release 1.2 - Programer's Guide*. Techinical Report no. 93-3, Dept. of. Information Science, University of Tokyo, 1993.

[13] S.Chiba. *OpenC++ Programmer's Guide for Version 2.*

[14] S.L.Corrêa, R.C. Silva, C.M.F. Rubira, L.E.Buzato. *Tolerância a Falhas de Software em Linguagens de Programação OO Reflexivas*. I SRTF, RS, Dezembro 1996 pp 57-64.

[15] F.Cristian. *Exception Handling*. Em Dependability of Resilient Computers, T.Anderson(Ed.),pp.68-97, BSP Professional Books, Oxford, 1989.

[16] F.Cristian. *Understanding Fault Tolerant Distributed Systems*. Communications of the ACM, New York, v.34,n.2,p.56-57, Fevereiro 1991.

[17] O . J. Dahl, B. Myhrhaug e K. Nygaar. *Simula 67 Common Base Language*. Publication no. s-22(revisada), Norwegian Computation Center, Oslo, October 1970.

[18] J.C.Fabre, V. Nicomette, T.Pérennou & Z.Wu. *Implementing Fault Tolerant Applications Using Reflective Object-Oriented Programming*. PDCS2 Second Year Report, Predictably Dependable Computing Systems, Newcastle upon Tyne, Inglaterra, Setembro 1994.

[19] J.Ferber. *Computational Reflection in Class-Based Object-Oriented Languages*. Em Proceedings of the 4th Annual Conference on Object-Oriented Programming: Systems, Languages and Application, OOPSLA'89, New Orleans, Lousiana, Special Issue of ACM SIGPLAN Notices, 24(10): 317-326, N.Meyrowitz (Ed.) Outubro 1989.

[20] B.Foote & R.E. Johnson. *Reflective Facilities in Smalltalk-80*. Em Proceedings of the 4th Annual Conference on Object-Oriented Programming:Systems, Language and Application, OOPSLA'99 New Orleans, Lousiana, Special Issue of ACM SIGPLAN Notices, 24(10): 327-335, N.Meyrowitz (Ed.) Outubro 1989.

[21] E. Gama, R. Helm, R.Johnson e J.Vlissides. *Design Pattern: Elements of Reusable Object Oriented Software*. Addison-Wesley Publishing, Massachusetts, USA, 1994.

[22] A . Goldberg & D.Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[23] R.E. Johnson e B.Foote. *Designing Reusable Classes*. Journal of Object Oriented Programming-JOOP, 1 (2):22-35, Junho/Julho 1988.

[24] G.Kiczales, J. des Rivieris & Bobrow. *The Art of the MetaObject Protocol*. MIT Press, 1991.

[25] A . Koenig & B. Stroustrup. *Exception Handling for C++*. Journal of Object-Oriented Programming 3(2):16-33, Jullho/Agosto 1990.

[26] P.A . Lee & T.Anderson. *Fault Tolerance: Principles and Pratice*. Second. Revised Edition, Springer-Verlag, 1990.

[27] M.L.Lisboa. *MOTF: Metaobjetos para Tolerância a Falhas*. Tese de doutorado, Instituto de Informática, UFRGS, Dezembro 1995.

[28] M.L.Lisboa, C.M.F.Rubira e L.E.Buzato. *Arquitetura Reflexiva para o Desenvolvimento de Software Tolerante a Falhas*. SEMISH'96, Recife, PE, agosto 1996, pp 155-166.

[29] P.H.C.Lisboa, J.F.Tepedino, S.L.Meira. *Reflexão Computacional em Smalltalk*. Revista Brasileira de Computação, Jul/Dez. 1993 Rio de Janeiro v.7 n.1 p.13-24.

[30] B.H. Liskov & A . Snyder. *Data Abstraction and Hierarchy*. Addendum to the Proceedings, OOPSLA'87, Special Issue of ACM SIGPLAN Notices, 23(5):17-34, Maio 1988.

[31] P.Madany, N.Islam, P.Kougioouris & R.H. Campbell. *Reification and Reflection in C++: An Operating Systems Perspective*, Techinical Report, no. UIUCDCs-R-92-1736, Department of Computer Science, University of Illinois at Urbana-Champaign, Março 1992 .

[32] P.Maes. *Concepts and Experiments in Computation Reflection* . Em Proceedings of the 2nd Annual Conference on Object-Oriented Programming: System, Language and Application, OOPSLA'87, Orlando, Florida, Special Issue of ACM SIGPLAN Notices, 22(12):147-155, N. Meyrowitz (Ed.), Outubro 1987.

[33] G.Masini, A . Napoli, D.Colnet, D.Leonard & K.Tombre. *Object-Oriented Languages. Academic* Press, 1991.

[34] M.Masuhara, S. Matsuoka, T.Watanabe, A . Yonezawa. *Object-Oriented Concurrent Reflective Languages can be implemented Efficiently*, OOPSLA'92 .

[35] S. Matsuoka, T. Watanabe, A . Yonezawa. *Hibrid Group Reflective Architeture for Object-Oriented Concurrent Reflective Programming*, ECOOP'91.

[36] P.Mulet, P.Cointe. *Definition of a Reflective Kernel for a Prototype Based Language*, Lectures Notes, Computer Science 1993.

[37] A . Paepcke. User-Level Language Crafting-Introducing the CLOS Metaobject Protocol. *Object-Oriented Programming - The CLOS Pespective*, capítulo 3. Andreas Paepcke (Ed.).

[38] D.P.Prado, S.L.Corrêa, C.M.F.Rubira. *Implementação de uma Pilha Robusta Tolerante a Falhas de Software em um Ambiente Distribuído*. I SRTF, RS, Dezembro 1996 pp 57-64.

[39] D.Powell. Delta4: *A Generic Architeture for Dependable Distributed Computing*, series Research Reports ESPRIT, 818/2252, Springer-Verlag, 1991.

[40] W. Pree. *Design Patterns of Object Oriented Software Development*. Addison-Wesley, 1995

[41] J.M.Purtilo e P.J.Jalote. *An Enviroment for Developing Fault-Tolerant Software*, IEE Transaction on Software Engineering, (17)2:153-159, February, 1991.

[42] B.Randell & J. Xu. *Object-Oriented Software Fault Tolerance: Framework, Reuse and Design Diversity*. PDCS2 First Year Report, Predictably Dependable Computing Systems, 1:165-184, Toulouse, France, Setembro 1993

[43] B.Randell, A .Romanovsk, C.Rubira, R.Stroud, Z.Wu, J.Xu. *From Recovery Blocks to Coordinated Atomic Actions*. Em Predictably Dependable Comp. Syst.. Eds. Randel et all, Springer-Verlag, pp.87-101 1995.

[44] C.M.F. Rubira & R.J.Stroud. *Foward and Backward Error Recovery in C++*. Journal of Object-Oriented Systems, 1(1):61-85, Outubro 1994.

[45] C.M.F.Rubira, S.L.Corrêa. *Um Framework Orientado a Objetos Reflexivo para a Construção de Software Tolerante a Falhas*. SCTF, RS, 1997.

[46] C.M.F.Rubira. *Structuring Fault-Tolerant Object Oriented Systems Using Inheritance and Delegation*. Ph.D. Thesis, Department of Compiting Science, University of Newcastle upon Tyne. Outubro 1994.

[47] J.Rumbaugh, M.Blaha, W.Premelani, F.Eddy & W.Lorense. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[48] H.A . Schmid. *Design Patterns for Constructing the host of a manufacturing framework*. Journal of Object-Oriented Programming-JOOP, Janeiro 1994

[49] S.K.Shrivastava, G.N.Dixon & G.D. Parrington. *An Overview of the Arjuna Distributed Programming System*. IEEE Software, 8(1):66-73, Janeiro 1991.

[50] R.C.Silva, S.L.Corrêa, C.M.F.Rubira, L.E.Buzato. *Reflexão Computacional em Linguagens de Programação: Um Estudo Comparativo*. II Simpósio Brasileiro de Linguagens de Programação SBLP'97, Campinas - SP, Setembro 1997.

[51] B.C.Smith. Prologue to *"Reflection and Semantics in a Procedural Language"*. Readings in Knowledge Representation edited by R.J.Brachman & Levisque, 1985.

[52] A . Snyder. *Encapsulation and Inheritance in Object-Oriented Programming Languages.* Em Proceedings of the 1st Annual Conference on Object-Oriented Programming: Systems, Languages and Application, OOPSLA'86, Portland, Oregon, Setembro/Outubro. Special Issue of ACM SIGPLAN Notice, 21(11):38-45, N. Meyrowitz (Ed.), Novembro 1986.

[53] R.J. Stroud. *Transparency and Reflection in Distributed Systems.* ACM Operating Systems Review, 22(2):99-103, Abril 1993.

[54] B.Stroustrup. *The C++ Programming Language.* Second Edition, Addison-Wesley, 1992.

[55] T.Takahashi, M.Takeda, *An Efficient Reflective Architeture for Parallel Logic Programming Languages*, Proceedings of ICLP'95 Workshop on Parallel Logic Programming.

[56] D. Ungar & R.B.Smith. Self: *The Power of the Simplicity.* Em Proceedings of the 2nd Annual Conference on Object-Oriented Programming: Systems, Languages and Applications, OOPSLA'87, Orlando, Florida, Special Issue of ACM SIGPLAN Notices, 22(12):227-241, N. Meyrowitz (ED.), Outubro 1987.

[57] K. Ushijima, S.Chiba e T.Masuda. *Meta-level programming for simpliflying library protocols.* ISOTAS'96, 1996.

[58] J. Xu, B. Randell, C.M.F. Rubira & R.J.Stroud. *Toward an Object-Oriented Approach to Software Fault Tolerance.* Em Fault-Tolerant Parallel and Distributed Systems, IEEE Computer Society Press, Outubro 1994.

[59] J.Xu, B.Randel, A . Romanovsky, C.M.F.Rubira, R.J.Stroud & Z.Wu. *Fault Tolerance in Concurrent OO Software Through Coordinated Error Recovery.* Proc.FTCS-25, Pesadena, CA, Junho 1995.

[60] Z.Yang, K.Dutty. *CORBA: A Plataform for Distribuited Object Computing ( A State of the Art Report on OMG/CORBA)* Operating Systems Reviews, Vol 30, number 2, pp 4-31, Abril 1996.

[61] Y.Yokote. *The Apertos Reflective Operating Systems: The Concept and Its Implementation.* Em Proceedings of the 7th Annual Conference on Object-Oriented Programming: Systems, Languages and Application, OOPSLA'92, Vancouver, British Columbia, Canada, Special Issue of ACM SIGPLAN Notices, 27(10): 414-434, A . Paepcke (Ed.) Outubro 1992.

[62] A . Yonezawa et. al.*Modelling and Programming in na OO Concurrent Language* ABCL/1. OO Concurrent Programming, MIT Press, 1987.

[63] P.Wegner. *Concepts and Paradigms of Object-Oriented Programming.* OOPS Messenger, 1(1):7-87, Agosto 1990.

[64] R.J. Wirfs-Brock e R.E. Johnson. *Surveying Current Research in Object-Oriented Design.* Communication of the ACM, 33(9), Setembro 1990.

[65] A . Zorzo, J.Xu & B.Randel. *Experimental Evaluation of Fault-Tolerant Mechanisms for OO Software.* SEMISH'96, Recife, PE, Agosto 1996, pp.457-468.