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

*Denise Piubeli Prado*

Dissertação de Mestrado

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

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Denise Piubeli Prado e aprovada pela Banca Examinadora.

Campinas, 29 de janeiro 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.

Instituto de Computação
Universidade Estadual de Campinas

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

Denise Piubeli Prado
Dezembro 1997

**Banca Examinadora:**

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

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

Profa. Dra. Eliane Martins
Instituto de Computação - UNICAMP

Prof. Dr. Edmundo Roberto Mauro Madeira (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. Eliane Martins

Prof. Dr. Cecília Mary Fischer Rubira

*Dedico este trabalho à memória de meu pai Pedro Prado e à toda a minha família pelo apoio, ajuda e incentivo imensamente dispensados.*

# Agradecimentos

À minha orientadora, Profa. Dra. Cecilia Mary Fischer Rubira, pelo apoio, incentivo, amizade, confiança e carinho dispensados ao longo da realização deste trabalho.

À Profa. Dra. Eliane Martins pela ajuda e paciência no estudo sobre Injeção de Falhas.

Aos meus familiares, especialmente pelos meus pais e meu marido, por sempre acreditarem e apoiarem o meu esforço.

A todos os meus amigos em especial pela amiga Sand Luz Corrêa pela ajuda e incentivo recebidos.

A CAPES pela concessão de uma bolsa de mestrado.

E finalmente a Deus.

O meu Muito Obrigada !!!

# Resumo

Este trabalho tem por objetivo desenvolver uma arquitetura orientada a objetos para dar suporte às aplicações tolerantes a falhas de software.

Técnicas de orientação a objetos, tais como, abstração de dados, herança, ligação dinâmica e polimorfismo são exploradas, visando obter aplicações de software de melhor confiabilidade e qualidade. Nosso objetivo é prover um suporte para aplicações que requeiram tolerância falhas de software através de técnicas já conhecidas de diversidade de projeto, integrando essas técnicas ao mecanismo de tratamento de exceções criando assim um framework composto por componentes de software genéricos que formam uma infra-estrutura para dar suporte ao desenvolvimento de sistemas tolerantes a falhas distribuídos(FOOD) .

# Abstract

The major goal of this work is to develop an object-oriented architecture for software fault-tolerant applications. Object-oriented techniques, such as data abstraction, inheritance and polymorphism are explored to improve software reliability and quality. Thus, our goal is to support software fault tolerance using design diversity, so that this support can be incorporated to the exception handling mechanism in the application. For the understanding and validation of these techniques, we have developed a fault-tolerant object-oriented distributed framework (FOOD).

# Conteúdo

# Lista de Figuras

# Lista de Tabelas

# Capítulo 1

# Introdução

Confiabilidade, segurança, distribuição, adaptabilidade, disponibilidade e tolerância a falhas são alguns dos requisitos que devem ser atendidos em aplicações de software modernas, como por exemplo, centrais telefônicas, bancos de dados, sistemas distribuídos, controle de trens e de tráfego aéreo. No entanto, o desenvolvimento de tais aplicações que atenda a todos esses requisitos simultaneamente não é uma tarefa simples, e requer o emprego de técnicas apropriadas durante todo o ciclo de desenvolvimento do software.

Um desses requisitos é a provisão de tolerância a falhas e está baseada na noção de redundância de componentes do sistema, tanto para a detecção de erros quanto para a recuperação dos mesmos. Entretanto, redundância implica diretamente em maiores custos de desenvolvimento dos sistemas e isso é um dos pontos cruciais que pesam na hora de sua escolha. Redundância pode ser dividida em duas categorias: redundância de hardware e redundância de software. Redundância de hardware é uma replicação de componentes físicos de hardware, sendo rotineiramente empregada para aumentar a disponibilidade/confiabilidade de sistemas de computação. A redundância de software é alcançada através de redundância de projetos de componentes de software; não é uma simples replicação. Isto implica que redundância de software é muito mais cara que redundância de hardware. As conseqüências diretas relacionadas ao uso de redundância de software são: (i) aumento no custo de desenvolvimento de sistemas e (ii) aumento da complexidade dos sistemas devido à introdução de componentes de software redundantes. Portanto, deve-se tomar muito cuidado com a introdução de redundância, de forma que ela seja incorporada no sistema de modo disciplinar e estruturado, pois senão o efeito é ao contrário, ou seja, os componentes redundantes podem diminuir, ao invés de aumentar, sua

**confiabilidade**. Chillarege[13] apresenta uma discussão detalhada sobre tolerância a falhas de software, enfocando principalmente onde e quando ela é necessária.

Devido ao alto grau de complexidade envolvido no desenvolvimento de um software, torna-se inevitável que ele possa ainda conter falhas de projetos residuais (tipicamente em erros lógicos) mesmo depois de passar pela fase de testes. Diante desse fato, técnicas e mecanismos são requeridos de maneira a habilitar um sistema a tolerar falhas de software que permaneceram no sistema.

Em geral, a maioria dos sistemas tolerantes a falhas se concentram em tolerar falhas de hardware, pelo fato dessas falhas serem mais facilmente detectadas e toleradas; entretanto, o principal problema reside em tolerar falhas de software, pois essas não são facilmente predizíveis, principalmente quando se trata em tolerá-las em ambientes distribuídos e paralelos.

Bloco de recuperação[46] e programação em n-versões[2] são duas técnicas de muito sucesso para a provisão de tolerância a falhas de software, embora não exista nenhum esquema que realmente possa garantir que todas as falhas de software sejam toleradas[38].

Outra noção importante envolvida no desenvolvimento de aplicações tolerantes a falhas é: **recuperação de erros por avanço** e **recuperação de erros por retrocesso**.

A distinção entre essas duas abordagens vem da possibilidade dos projetistas em prever falhas que podem ocorrer durante a execução do sistema, ou seja, se as falhas que ocorreram durante a execução do sistema foram previstas, então a abordagem adequada é a recuperação de erros por avanço, senão, se as falhas que ocorreram durante a execução do sistema não foram previstas, então a abordagem adequada é a recuperação de erros por retrocesso.

Exceções[15,25] e tratamento de exceções são os mecanismos mais comuns para prover recuperação de erros por avanço. Para a técnica de recuperação de erros por retrocesso, a única solução é substituir o estado completo do sistema, restaurando-o a um estado anterior à manifestação da falha. Diversas linguagens de programação dão suporte para a recuperação de erros por avanço utilizando algum mecanismo de tratamento de exceções. Entretanto, poucos estudos têm sido realizados para dar suporte à recuperação de erros por avanço em ambientes distribuídos. Já para a recuperação de erros por retrocesso, alguns ambientes transacionais, como por exemplo Arjuna[22,55] possuem facilidades para o seu suporte em um ambiente distribuído. Contudo, a provisão de ambos os mecanismos em um mesmo ambiente de programação tem sido muito pouco explorada.

O modelo adotado nessa dissertação para a implementação de sistemas tolerantes a falhas distribuídos é o paradigma de objetos. A programação orientada a objetos é considerada pela comunidade científica como um modelo de programação efetivo para a produção de software de melhor qualidade devido aos seus benefícios, tais como modularidade, abstração de dados, encapsulamento, hierarquia, polimorfismo e reutilização de código. Em geral, o uso de técnicas orientadas a objetos para implementar os mecanismos de tolerância a falhas facilita o controle da complexidade do sistema pois elas promovem uma melhor estruturação de seus componentes. Além disso, esses componentes podem ser reutilizados em uma grande variedade de aplicações similares, barateando o custo de desenvolvimento de software tolerante a falhas.

O objetivo desta dissertação é propor um framework composto por componentes de software genéricos que formam uma infra-estrutura que dê suporte para o desenvolvimento de sistemas tolerantes a falhas. Framework, segundo Johnson[65] é um conjunto de classes concretas e abstratas, bem como suas inter-conexões, que provê uma infra-estrutura genérica de soluções para um conjunto de problemas e também um grande alcance para reutilização de software em grande escala. A definição de framework é útil para a construção de sistemas tolerantes a falhas no sentido de promover não apenas a reutilização de componentes existentes, mas também promover transparência aos programadores da aplicação, que apenas utilizam seus serviços sem se preocuparem como eles são

implementados. Essas vantagens apresentadas mostram que a construção de frameworks, principalmente através da diversidade de classes, unida ao paradigma de objetos é extremamente importante para o desenvolvimento de aplicações tolerantes a falhas.

O framework foi desenvolvido em C++[26,33,51,58] usando o ambiente para programação distribuída Arjuna que dá apoio ao mecanismo de recuperação de erros por retrocesso. Para demonstrar a aplicabilidade e viabilidade da abordagem proposta, um pequeno estudo de caso foi implementado.

## 1.1 Contribuições

As contribuições mais importantes dessa dissertação, são as seguintes:

⇒ Implementação de um framework em C++ que dá apoio ao desenvolvimento de aplicações tolerantes a falhas de software;

⇒ Incorporação do mecanismo de tratamento de exceções em um ambiente distribuído.

## 1.2 Organização do texto

O Capítulo 2 provê informações básicas e terminologias relacionadas ao paradigma de objetos. O Capítulo 3 descreve alguns conceitos fundamentais de tolerância a falhas. O Capítulo 4 apresenta o projeto e implementação do framework e as classes que o compõe. Finalmento o Capítulo 5 apresenta as conclusões deste trabalho e algumas sugestões de possíveis extensões.

# Capítulo 2

# Fundamentos de Orientação a Objetos

Este capítulo apresenta um conjunto de técnicas básicas que são centrais para a filosofia de programação orientada a objetos, e que foram muito utilizadas no desenvolvimento desta dissertação. O objetivo é prover definições claras de noções fundamentais como por exemplo, tipos abstratos de dados, objetos, classes, herança, ligação dinâmica e polimorfismo. As notações apresentadas seguem a metodologia para projeto orientado a objetos OMT[53].

A estrutura deste capítulo é organizada como segue: a Seção 2.1 discute os conceitos básicos sobre programação orientada a objetos, tais como abstração, objetos, classes, herança, polimorfismo e ligação dinâmica; a Seção 2.2 apresenta uma metodologia orientada a objetos para o projeto orientado a objetos usada nessa dissertação e finalmente a Seção 2.3 apresenta um resumo deste capítulo.

## 2.1 Conceitos Básicos

### 2.1.1 Abstração

Abstração[12,53] consiste na concentração dos aspectos essenciais para a descrição de uma entidade, ignorando suas propriedades irrelevantes. O uso da abstração durante o processo de análise significa lidar apenas com conceitos do domínio da aplicação, e não tomar decisões adiantadas sobre o projeto e implementação antes do problema ser completamente compreendido.

Três conceitos importantes podem ser abstraídos na observação de um fenômeno: (i) uma categoria ou classe, (ii) uma ação e (iii) um atributo. Baseado nessas noções, três operações básicas podem ser identificadas que envolvem abstração, onde cada uma possui sua operação inversa: classificação/instanciação; generalização/especialização e agregação/decomposição, que serão discutidas abaixo.

- **Classificação**: é o relacionamento que descreve um número de objetos que estão considerados em uma categoria devido as suas características similares. Por exemplo, duas pessoas, Carlos e Celina são classificados como instâncias de Pessoa. Classificação pode ser descrita como um relacionamento "é um", por exemplo, Carlos é uma Pessoa.

- **Agregação**: é o relacionamento "parte-todo" ou "uma-parte-de" no qual os objetos que representam os componentes de alguma entidade são associados a um objeto que representa a estrutura inteira, reduzindo de maneira significante a complexidade por tratarmos muitos objetos como um único objeto. Por exemplo, um nome, uma lista de argumentos e uma declaração composta fazem parte da definição de funções da linguagem C, definição que, por sua vez, faz parte de um programa. A agregação é uma forma de associação com algumas restrições a mais.

- **Generalização**: é o relacionamento simples entre uma classe, e uma ou mais versões refinadas dela. A classe que estiver em processo de refinamento é chamada de superclasse e cada versão refinada dela é chamada de subclasse. Por exemplo, estudantes de graduação e pós-graduação podem ser considerados subclasses da classe estudante. Estudante é uma generalização de estudante graduado e pós-graduado.

Pode-se observar que agregação e generalização são noções independentes. Em outras palavras, elas são ortogonais e quando empregadas juntas constituem meios para obter uma rica ferramenta para modelagem.

### 2.1.2 Tipos Abstratos de Dados

Tipos abstratos de dados[12,24,53] estão relacionados com o princípio de abstração de dados através da separação da especificação da abstração de dados de sua implementação. Isto proporcionou uma grande melhoria devido à abstração ser diretamente refletida na sintaxe e na semântica de uma linguagem de programação. Tipos abstratos de dados são pioneiros em linguagens de programação como por exemplo, Clu[40] e Módula-3[52], e mais recentemente como característica principal de Ada[54].

Tipos abstratos de dados são constituídos de duas partes: a parte da especificação e a parte da implementação. Cada parte pode também ser, por sua vez, subdividida: a especificação sendo denotada pela sintaxe e semântica; e a implementação pela representação e algoritmos. A Figura 2.1 representa esta divisão:

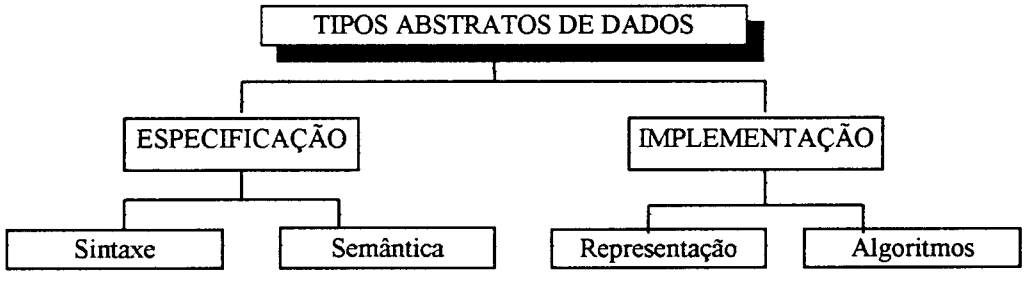



Figura 2.1: Tipos Abstratos de dados

A principal vantagem dessa separação consiste no fato de permitir o uso de um tipo sem necessariamente saber sobre a sua implementação. A parte da especificação de um tipo abstrato de dado especifica a sintaxe de um tipo abstrato de dado e sua semântica. A sintaxe consiste na assinatura das operações que definem a interface, sem no entanto, ser suficiente para descrever a sua semântica. A implementação de um tipo abstrato de dado descreve sua representação em termos de estruturas de dados primitivas (representação) e os algoritmos associados a cada operação (algoritmos).

### 2.1.3 Objetos

Objeto é uma entidade que encapsula informação de estado e dados, e um conjunto de operações associadas que manipulam os dados (Figura 2.2). Em geral, cada estado do objeto é completamente protegido e oculto dos outros objetos, e o único modo de examiná-lo é através de invocações das operações acessíveis publicamente. Cada objeto tem uma identidade única distinguindo-o dos demais objetos.

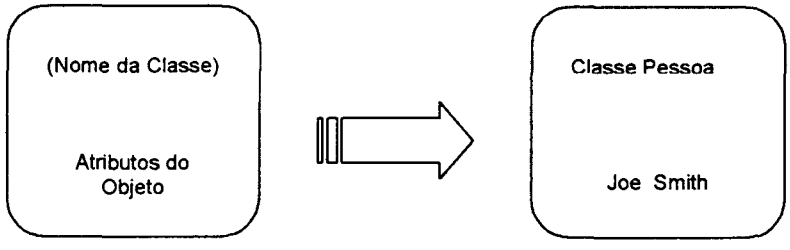



Figura 2.2: Notação para Objetos

### 2.1.4 Classes

Uma classe[3,8,53] especifica propriedades e comportamentos para um conjunto de objetos similares, ou seja, ela descreve um grupo de objetos com propriedades semelhantes (atributos), o mesmo comportamento (operações ou métodos), os mesmos relacionamentos com outros objetos e a mesma semântica. Método corresponde à implementação de uma operação para uma classe e mensagem corresponde à assinatura de uma operação. A interface pública é definida pelo conjunto de métodos e mensagens que podem ser requisitados por outros objetos; a visão externa de um objeto nada mais é do que a sua interface pública. Uma classe é representada por um retângulo de linhas sólidas podendo

haver três compartimentos: o nome da classe, uma lista de atributos e uma lista de operações, como mostra a Figura 2.3.

Os princípios de abstração de dados, encapsulamento e modularidade são alcançados na programação orientada a objetos através do uso de conceitos de classes e objetos.

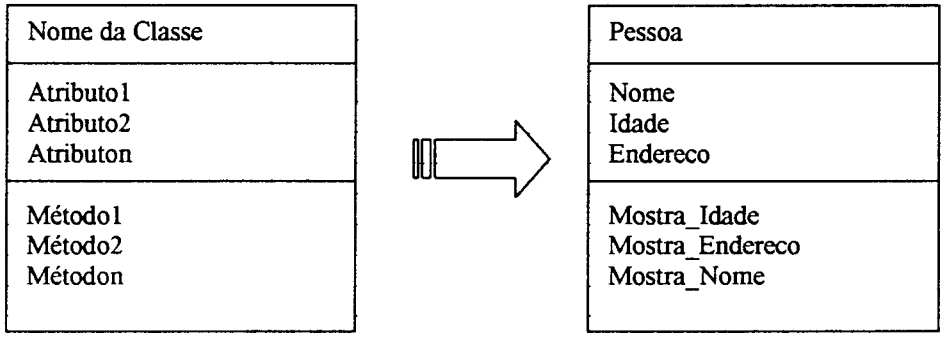



Figura 2.3: Notação para classes

### 2.1.5 Tipos

Tipos, segundo Ghezzi e Jazayeri[24], podem ser visualizados como uma especificação de um conjunto de valores que podem ser associados com uma variável, juntamente com as operações que podem ser legalmente usadas para criar, acessar, e modificar tais valores. Por exemplo, o tipo Booleano limita-se a uma certa classe de valores Verdadeiro e Falso e para as operações lógicas E, Ou e Negação. No modelo de objetos, tipos podem ser definidos como uma coleção de objetos que respondem do mesmo modo a um mesmo conjunto de mensagens, ou seja, um tipo é uma coleção de objetos que respondem a uma mesma interface pública.

Não se pode confundir tipo com classe. Tipo é essencialmente a descrição de uma interface, enquanto que classe especifica a implementação particular de um tipo.

### 2.1.6 Encapsulamento

O encapsulamento[8,53], também chamado de ocultamento de informações, consiste na separação dos aspectos externos de um objeto, acessíveis por outros objetos, dos detalhes internos da implementação daquele objeto, que ficam ocultos dos demais objetos. O encapsulamento minimiza interdependências entre módulos através de interfaces externas bem definidas. O uso do encapsulamento não é exclusivo das linguagens orientadas a objetos, porém a capacidade de combinar estrutura de dados e seu comportamento em uma única entidade torna-o mais completo e mais poderoso do que nas linguagens convencionais, que separam a estrutura de dados de seu comportamento.

Por exemplo, em uma linguagem orientada a objetos típica, a definição de uma classe é uma entidade cuja interface externa consiste de um conjunto de operações; mudanças na implementação de uma classe que preserva a interface externa não afeta o código fora da definição da classe. O mecanismo de classe permite ao programador encapsular sintaticamente a descrição de estrutura de dados junto com a interface pública. Entretanto, o mecanismo de herança introduz uma nova categoria de cliente para uma classe: as subclasses. Uma classe tem, portanto, dois tipos de clientes :

- **Clientes por Instanciação**: Aqueles que somente criam instâncias da classe e manipulam essas instâncias através de operações associadas com a classe;

- **Clientes por Herança:** Aqueles que são subclasses e herdam operações e estrutura da classe.

Nem todas as linguagens orientadas a objetos implementam as mesmas regras em relação ao acesso e visibilidade a esses dois tipos de clientes. A maioria das linguagens dão suporte às noções de público, privado e visível pela subclasse, deixando para o programador a especificação da proteção desejada. Abaixo segue a definição das três opções:

- **Pública**: Se um atributo ou uma operação são declarados como públicos, então o cliente pode diretamente acessar, manipular ou invocá-los;
- **Privada** : Se um atributo ou uma operação são declarados como privados, então nenhum cliente pode diretamente acessar, manipular ou invocá-los;
- **Visível pela Subclasse**: Se um atributo ou uma operação são declarados serem do tipo visível pela subclasse, então eles podem ser acessados, manipulados, ou invocados diretamente apenas pelos clientes herdados, ou seja, pelas subclasses.

Na linguagem de programação orientada a objetos C++[26,58], as palavras reservadas *public*, *private* e *protected* (visível pela subclasse) são usadas, respectivamente, para prover a visibilidade das operações e atributos.

### 2.1.7 Herança

O mecanismo de herança[3,8,53] permite-nos definir novas classes a partir da extensão de classes já existentes, ou seja, permite a criação de subclasses ou classes derivadas. A classe já existente é chamada de classe base.

Em geral, a classe derivada herda todos os membros de classe (atributos e operações) de sua classe base, podendo também adicionar novas operações, estender a representação de dados ou ainda rescrever a implementação de operações existentes. Se a classe base, ainda, é derivada de outra classe, os membros desta classe são também herdados. Classes são freqüentemente organizadas em uma hierarquia (uma estrutura de árvore) e uma classe herda de todos os seus antecessores nesta hierarquia. De acordo com Liskov[41], podemos obter dois tipos de hierarquias através do uso de herança: hierarquia de tipo e hierarquia de implementação. Uma hierarquia de tipo, é composta de 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 do tipo S pode então ser usado como se ele fosse do tipo T porque é garantido prover pelo menos as operações de T. Ou seja, um objeto do tipo S pode substituir um objeto do tipo T. A questão chave é que as classes derivadas se comportem

como se fossem as classes pais(bases). O uso de herança relaciona o comportamento de dois tipos, e não necessariamente suas implementações.

Hierarquia de implementação é quando o mecanismo de herança pode ser usado como uma técnica para implementar tipos abstratos de dados que são similares a outros tipos já existentes. Nesse caso, a intenção é garantir que a subclasse herde as características que são similares da classe base.

### 2.1.8 Ligação Dinâmica

Em geral, o conceito de ligação reflete uma associação entre um atributo e uma entidade. Exemplos de entidades de programas são variáveis, subprogramas e comandos. Os atributos de uma variável são seu nome, valor, tipo e área de armazenamento[24]. Uma ligação de uma variável é estática se ela ocorre antes do tempo de execução[1] e permanece inalterada através da execução do programa. Se a ligação ocorre durante o tempo de execução ou muda no curso da execução do programa, então ela é chamada de ligação dinâmica. A maior vantagem de ligação dinâmica é que ligações podem ser mudadas no decorrer do programa, gerando muita flexibilidade de programação. Em programação orientada a objetos, ligação dinâmica refere-se ao mapeamento entre o nome da operação e sua implementação, ou seja, uma mensagem invocada em tempo de execução é associada dinamicamente a uma implementação que depende da classe do objeto que invocou a mensagem. Na linguagem de programação C++, o programador deve requisitar uma ligação dinâmica para uma mensagem explicitamente declarando-a como virtual na classe base e redefinindo-a na classe derivada. Se o tipo do objeto que invocar essa operação for da classe base, então ele será associado à implementação definida pela classe base; se o tipo do objeto for do tipo da classe derivada, então a operação invocada será associada à implementação da classe derivada.

[1] Do inglês: *run-time*

### 2.1.9 Polimorfismo

Polimorfismo[12,18,53] é definido como sendo a habilidade que algumas variáveis têm de poder referenciar mais de um tipo, possibilitando o seu uso em diferentes contextos que exigem diferentes tipos. Polimorfismo não é uma idéia nova. Diversas formas de polimorfismo já existiam em linguagens de programação há muito tempo. Cardelli e Wegner[12] propõe uma classificação para polimorfismo em duas categorias: *ad hoc* e universal (Figura 2.4). Coerções e sobrecarga são duas formas de polimorfismo *ad hoc* apoiadas por muitas linguagens de programação, como por exemplo, Algol68 e Ada:

- **Coerções** provêm um modo simples de contornar a rigidez de linguagens monomórficas através de uma forma limitada de polimorfismo. Linguagens que apoiam coerções têm certos mapeamentos (coerções) construídos entre tipos. Se um contexto particular requer um tipo específico e um tipo diferente é provido pelo programador, então a linguagem irá procurar ver se há uma coerção apropriada. Por exemplo, se a operação de adição é definida com dois reais e um inteiro, e apenas números reais são fornecidos como parâmetros, então o inteiro irá ser convertido para um número real através de uma coerção;

- **Sobrecarga**: permite que um nome de função seja usado mais que uma vez com diferentes tipos de parâmetros. Por exemplo, a função adiciona poderia ser usada para operar com ambos números inteiros e reais. A informação do tipo de parâmetro irá ser usada para definir qual função será executada.

A Figura abaixo ilustra a taxonomia usada por Cardelli e Wegner:

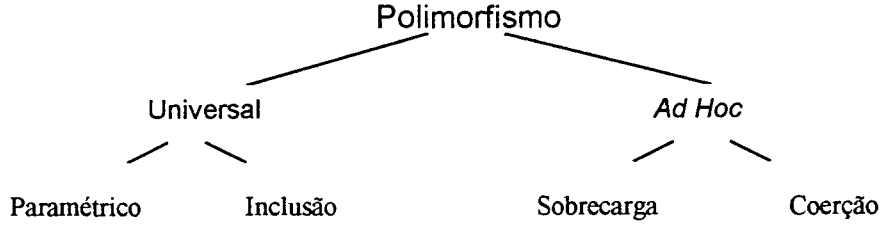



Figura 2.4: Taxonomia de Cardelli e Wegner

No polimorfismo *ad hoc*, ou seja, coerção e sobrecarga, não existe uma maneira única para determinar o tipo do resultado de uma função com base no tipo de seus argumentos, por isso esse tipo de polimorfismo trabalha com um número específico de tipos, diferindo exatamente neste ponto do polimorfismo universal que trabalha com um conjunto infinito de tipos. Para melhor esclarecer, considere o seguinte exemplo: considere uma função que irá retornar o tamanho (em bytes) de uma estrutura de dados em particular. Em uma linguagem com polimorfismo universal, isto seria especificado apenas uma vez de um modo geral e operaria sobre todos os tipos; já no polimorfismo *ad hoc*, mais diretamente através da técnica de sobrecarga, a função seria escrita N vezes para N tipos diferentes, isto significa que a função seria restrita aos tipos pertencentes ao conjunto dos N tipos.

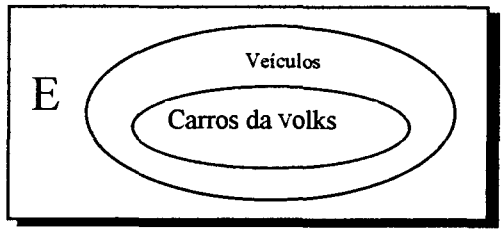
O polimorfismo universal é composto de duas técnicas:

- **Paramétrico**: neste tipo de polimorfismo[18], uma operação única irá trabalhar uniformemente sobre um conjunto de tipos. Funções paramétricas são também chamadas de funções genéricas, ou seja, funções que trabalham genericamente sobre um conjunto de tipos, de modo a parametrizar um componente de software com um ou mais tipos. Uma classe genérica ou parametrizada é uma classe que serve como um "template" para outras classes; o "template" pode ser parametrizado por outras classes, objetos e/ou operações. Uma classe genérica deve ser "instanciada" antes que seus objetos sejam criados.

- **Inclusão**: o polimorfismo de inclusão também permite que uma operação trabalhe sobre um conjunto de tipos, entretanto, esse conjunto é determinado pelos relacionamentos de subtipo. Com o polimorfismo de inclusão, uma função definida para um tipo particular pode também operar sobre qualquer subtipo daquele tipo.

O conceito de subtipo abrange linguagens orientadas a objetos estaticamente tipadas. Por exemplo, seja um tipo que representa os carros da Volks, que é um subtipo de um tipo mais geral chamado Veículo (Figura 2.5). Todo objeto de um subtipo pode ser

usado em um contexto de supertipo, ou seja, todo carro da Volks é um veículo e pode ser operado pelas operações aplicáveis aos veículos.




Veículos

Figura 2.5: Exemplo de polimorfismo de inclusão

As linguagens polimórficas têm diversas vantagens sobre as monomórficas, como por exemplo, flexibilidade, abstração, compartilhamento e código, responsáveis pelo entendimento de linguagens orientadas a objetos tipadas.

### 2.1.10 Classes Concretas e Abstratas

Uma classe abstrata[53] é uma classe que não possui instâncias diretas mas suas classes descendentes têm instâncias diretas. Uma classe concreta é uma classe que é instanciável, ou seja, possui instâncias diretas. Uma classe concreta pode ter subclasses abstratas (mas estas, por sua vez, devem possuir descendentes concretos). Somente classes concretas podem ser "classes de folhas" na árvore de herança, ou seja, somente classes concretas podem ocupar o último nível de uma árvore de herança. A Figura 2.6 ilustra a definição da classe abstrata Empregado.

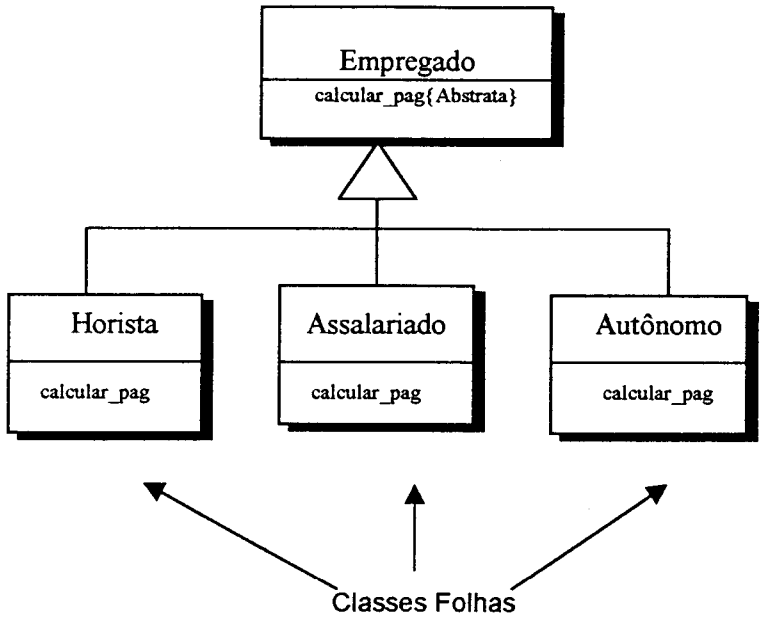



Figura 2.6: Exemplo de classe abstrata

As classes abstratas, segundo Rumbaugh[53], organizam características comuns entre diversas classes. Muitas vezes é útil criar uma superclasse abstrata para encapsular classes que participam da mesma associação ou agregação. Algumas classes abstratas surgem naturalmente no domínio da aplicação, outras são introduzidas artificialmente como um mecanismo para facilitar a reutilização de código. O conceito de framework, apresentado no Capítulo 1, é baseado em classes abstratas de forma a prover um mecanismo para reutilização de código.

As classes abstratas são freqüentemente usadas para definir operações a serem herdadas pelas subclasses. Por outro lado, uma classe abstrata pode definir o protocolo para uma operação sem apresentar uma implementação correspondente. Isso é chamado de operação abstrata. Uma operação abstrata define a forma de uma operação para a qual cada subclasse concreta deve prover sua própria implementação. Na Figura 2.6, suponha que a classe abstrata Empregado tenha uma operação abstrata chamada **calcular_ pag**, onde a operação é definida, mas não sua implementação. Neste caso, cada subclasse deve prover um método (ou uma implementação) para esta operação.

Observe que a natureza abstrata de uma classe é sempre provisória, dependendo do seu uso. Uma classe concreta normalmente pode ser refinada em várias subclasses, podendo se tornar abstrata.

## 2.2 Metodologia OMT

A técnica de modelagem de objetos OMT [53] é uma notação gráfica independente de linguagem, que representa os principais conceitos de orientação a objetos. A metodologia apresenta as etapas do ciclo de vida de um sistema que são: análise, projeto de sistema, projeto dos objetos e implementação.

Esta técnica faz uso de três tipos de modelos para descrever a fase de análise de uma aplicação orientada a objetos (Figura 2.7), que são: (i) o **modelo de objetos**, que descreve os objetos do sistema e seus relacionamentos; (ii) o **modelo dinâmico**, que descreve as interações entre os objetos do sistema; e (iii) o **modelo funcional**, que descreve as transformações de dados do sistema.

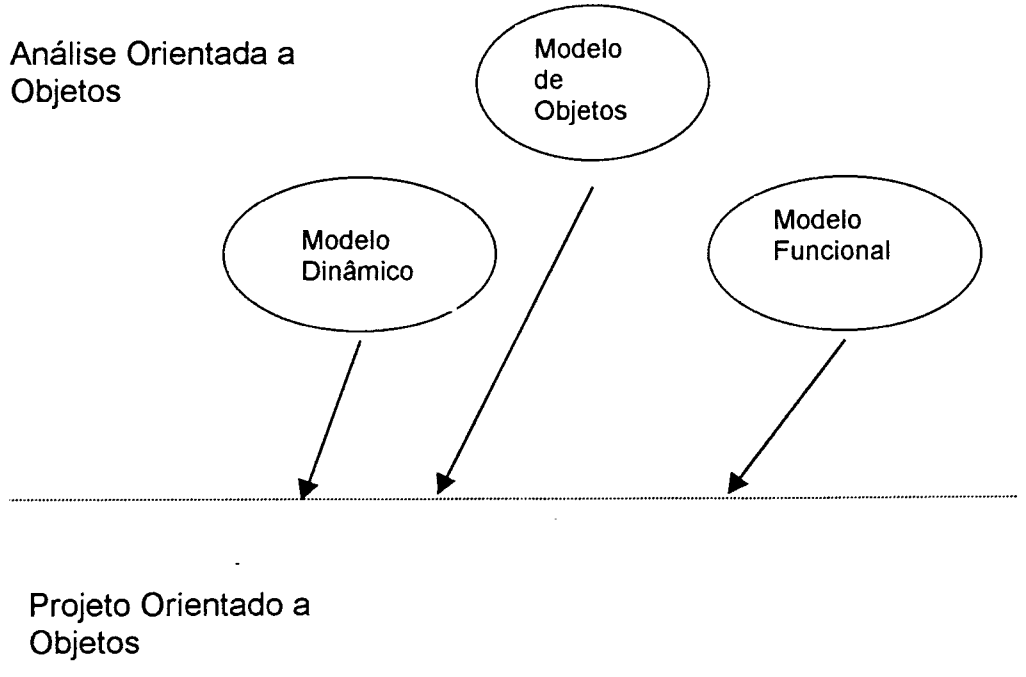



Figura 2.7: Resumo do Processo de Análise orientada a Objetos

Os três modelos são partes ortogonais da descrição da aplicação. O modelo de objetos é o mais importante por ser mais necessário descrever **o que** está mudando ou se transformando antes de descrever **quando** ou **como** isso acontece. Cada um dos três modelos evolui durante o ciclo de desenvolvimento de um sistema. Durante a análise é construído um modelo de domínio da aplicação independentemente de uma possível implementação. Durante o projeto, são adicionadas ao modelo construções para o domínio da solução. Durante a implementação, são codificadas tanto abstrações do domínio da aplicação quanto do domínio da solução.

### 2.2.1 Modelo de Objetos

O modelo de objetos[53] descreve a estrutura de objetos de um sistema, seus relacionamentos com outros objetos, seus atributos e suas operações. O modelo de objetos proporciona a estrutura necessária na qual podem ser acoplados os modelos dinâmico e funcional. Modificações e transformações não fazem sentido a menos que haja alguma coisa para ser modificada. Na modelagem de um problema de engenharia, por exemplo, o modelo de objetos deve conter termos familiares a engenheiros; na modelagem de um problema comercial, termos da área de negócios; na modelagem de uma interface com o usuário, termos do domínio da aplicação. O modelo da análise não deve conter construções computacionais a menos que a aplicação que esteja sendo modelada seja intrinsecamente um problema computacional tal como um compilador ou um sistema operacional.

O modelo de objetos é representado graficamente por diagramas de objetos contendo classes de objetos. As classes são organizadas em níveis hierárquicos compartilhando estruturas e comportamentos comuns e são associadas a outras classes. As classes definem os valores de atributos relativos à cada instância de objetos e às operações que cada objeto executa ou a que se submete. O modelo de objetos pode simplesmente ser resumido da seguinte maneira:

| Modelo de Objetos = Diagrama de Objetos + Dicionário de Dados |
|---|

### 2.2.2 Modelo Dinâmico

O modelo dinâmico[53] descreve os aspectos de um sistema relacionados ao tempo e à seqüência de operações - eventos que assinalam modificações, seqüências de eventos, estados que definem o contexto para os eventos, e a organização de eventos e estados. O modelo dinâmico incorpora o controle, que é um aspecto de um sistema que descreve as seqüências de operações que ocorrem, independentemente do que as operações fazem, sobre o que elas atuam ou como são implementadas.

O modelo dinâmico é representado graficamente por diagramas de estados. Cada um desses diagramas mostra a seqüência de estados e eventos permitidos em um sistema para uma classe de objetos. Os diagramas de estados também se relacionam com os outros modelos. As ações nos diagramas de estados correspondem à funções do modelo funcional; os eventos , por sua vez, às operações em objetos no modelo de objetos. O modelo dinâmico pode ser resumido da seguinte maneira:

| Modelo Dinâmico = Diag. de Estados + Diag. de Fluxo de Eventos Globais |
|---|

### 2.2.3 Modelo Funcional

O modelo funcional[53] descreve os aspectos de um sistema relacionados a transformações de valores: funções, mapeamentos, restrições e dependências funcionais. O modelo funcional abrange o que um sistema faz, independentemente de como ou quando é feito. O modelo funcional é representado por meio de diagramas de fluxos de dados. Esses diagramas mostram as dependências entre valores e o processamento dos valores de saída a partir dos valores de entrada e das funções, independentemente de quando ou se as funções são executadas. Os conceitos tradicionais da computação como árvores de expressões são exemplos de modelos funcionais, bem como conceitos menos tradicionais como planilhas eletrônicas. As funções são chamadas como ações no modelo dinâmico e mostradas como

operações sobre os objetos no modelo de objetos. O modelo funcional pode ser resumido da seguinte maneira:

| Modelo Funcional = Diagrama de Fluxo de Dados + Restrições |
|---|

### 2.2.4 Relacionamento entre os modelos

Cada modelo individualmente descreve um aspecto do sistema mas contém referências aos outros modelos. O modelo de objetos descreve a estrutura de dados sobre a qual atuam os modelos dinâmico e funcional. As operações do modelo de objetos correspondem aos eventos do modelo dinâmico e às funções do modelo funcional. O modelo dinâmico descreve a estrutura de controle de objetos, ele mostra as decisões que dependem dos valores dos objetos e que provocam ações que modificam esses valores e que chamam funções. O modelo funcional descreve as funções chamadas pelas operações do modelo de objetos e pelas ações no modelo dinâmico. As funções operam sobre os valores de dados especificados pelo modelo de objetos. O modelo funcional também mostra restrições relativas aos valores dos objetos.

## 2.3 Sumário

Neste capítulo conceitos como abstração, tipos abstratos de dados, objeto, classe, herança, encapsulamento, polimorfismo e ligação dinâmica foram definidos e foi mostrado que o paradigma de objetos não é somente um estilo de programação mas também um método de projeto para a construção de sistemas.

Apresentamos também a metodologia de projeto orientada a objetos OMT destinada ao desenvolvimento de aplicações orientadas a objetos e que combina três visões de modelos para a fase de análise de um sistema: modelo de objetos, modelo funcional e modelo dinâmico.

# CAPÍTULO 3

# Fundamentos de Tolerância a Falhas

Este capítulo apresenta um resumo dos conceitos principais de tolerância a falhas usados nessa dissertação. Ele irá apresentar conceitos relacionados a componentes, sistemas, redundância de software e de hardware, técnicas de tolerância a falhas de software e de hardware. O capítulo é dividido da seguinte forma: a Seção 3.1 apresenta alguns conceitos básicos sobre tolerância a falhas; a Seção 3.2 discute as técnicas de tolerância a falhas de software; a Seção 3.3 introduz tolerância a falhas de hardware orientada a objetos; a Seção 3.4 discute tolerância a falhas de software orientada a objetos e por fim a Seção 3.5 apresenta o resumo deste capítulo.

## 3.1 Conceitos de Tolerância a Falhas

### 3.1.1 Sistema

Antes de definirmos os conceitos sobre tolerância a falhas, é apropriado que uma definição mais precisa dos termos **sistema** e **modelo de sistema** seja dada.

Um sistema segundo Lee and Anderson[38] é qualquer mecanismo identificável que mantém um padrão de comportamento diante de uma interface entre o mecanismo e o ambiente. Um sistema consiste de um conjunto de **componentes** que se interagem sob o controle de um projeto que é também um componente do sistema [1,38]. A Figura 3.1 representa graficamente este conceito. Um modelo de sistema trata da estrutura interna dos sistemas de maneira a estipular o modo como os sistemas são construídos a partir de seus

componentes. O modelo do sistema é recursivo no sentido que cada componente é considerado sozinho como um sistema, que por sua vez, pode conter outro sistema. Os componentes recebem pedidos para realizar um serviço e produzem respostas. Se um componente não puder satisfazer um pedido para um determinado serviço, ele produzirá uma resposta de falha.

Um sistema tolerante a falhas é aquele capaz de continuar a operar mesmo após a ocorrência de uma falha.

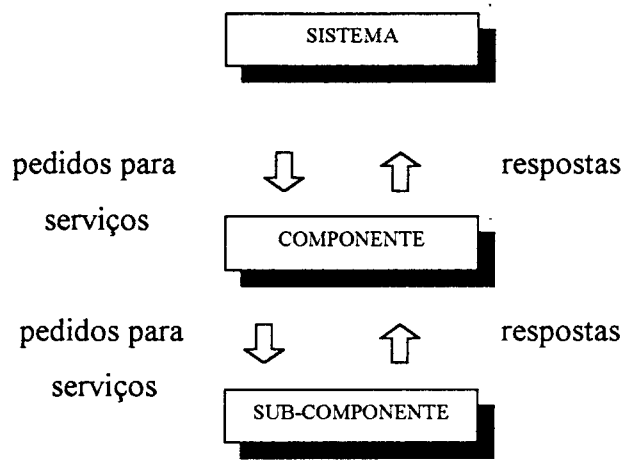



Figura 3.1 Conjunto de Componentes de um Sistema

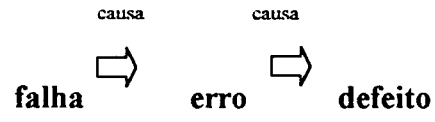
### 3.1.2 Falha, Erro e Defeito

**Falha**[2] segundo [1,38] é definida como sendo um defeito no estado interno de um componente ou projeto podendo levar a um erro. Uma falha de componente em um sistema leva a um erro no estado interno do componente; uma falha de projeto em um sistema leva a um erro no estado do projeto. Um **erro**[3] é um estado não válido do sistema

[2]Do inglês: Fault
[3] Do inglês: Error

que pode, por sua vez, levar a um **defeito**[4] . Finalmente, o defeito de um sistema ocorre quando este desvia-se daquilo que lhe foi especificado.

No contexto de tolerância a falhas, esses conceitos são resumidos da seguinte maneira:




A meta de tolerância a falhas é justamente prever falhas e erros antes que isso possa levar ao fracasso do sistema. Dois tipos de falhas devem ser consideradas no desenvolvimento de técnicas de tolerância a falhas: falhas de hardware e falhas de software. Falhas de hardware são aquelas causadas por componentes físicos no computador e falhas de software são aquelas causadas por erros lógicos deixados no software durante o seu desenvolvimento. Essa dissertação concentra-se em tolerância a falhas de software.

### 3.1.3 Componente Ideal

Como discutido na seção 3.1.1, um sistema consiste de um conjunto de componentes que interagem sob o controle de um projeto, que também é um componente do sistema. Componentes por sua vez também podem ter sub-componentes. Componentes e sub-componentes recebem requisições de serviços e produzem respostas a essas requisições (Figura 3.1).

As respostas aos pedidos de componentes podem ser classificadas em duas categorias: **normais** e **anormais**. Respostas normais são aquelas que um componente (ou sub-componente) produz quando tudo ocorre como especificado e as respostas anormais são aquelas produzidas quando o comportamento do componente se desvia do especificado. As respostas anormais de um componente(ou sub-componente) são chamadas de exceções e significam que alguma situação excepcional ocorreu neste componente. Como

[4] Do inglês: failure

conseqüência dessa divisão de respostas em duas categorias, a atividade de um componente também pode ser dividida em **atividade normal** e **atividade anormal** (tratamento de exceções). Essa separação leva-nos à definição do **componente tolerante a falhas ideal**[37,38]. A Figura 3.2 define esse componente.

Três classes de situações excepcionais (onde a tolerância a falhas é necessária) são identificadas:

- **Exceções de Interface**: São transmitidas quando verificações na interface encontram um pedido de serviço inválido sendo feito a um componente. Estas exceções devem ser tratadas pela parte do sistema que fez o pedido inválido;
- **Exceções Locais**: Essas exceções são sinalizadas quando um componente detecta um erro que suas próprias capacidades de tolerância a falhas poderiam ou deveriam tratar na esperança que o componente retorne as suas operações normais depois do tratamento de exceções;
- **Exceções de Falha**: São transmitidas para notificar ao componente que fez o pedido de serviço que, além do uso de suas capacidades de tolerância a falhas, ele não foi capaz de executar o pedido.

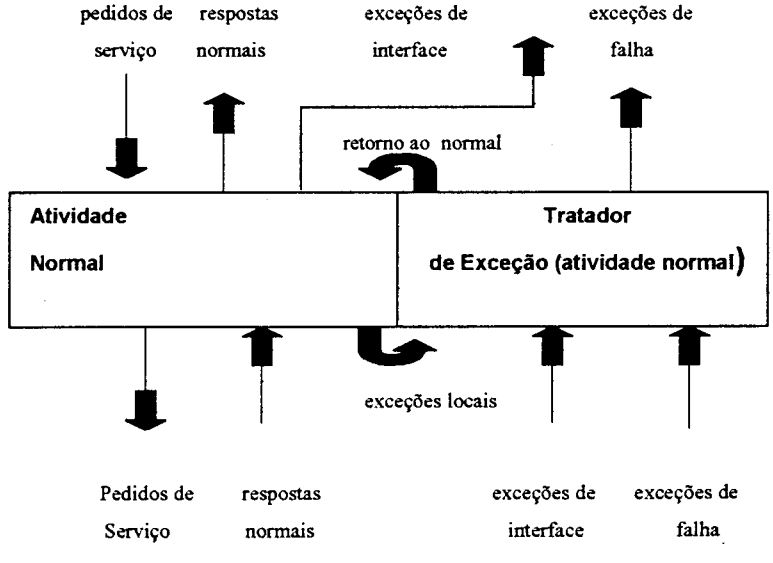



Figura 3.2 Componente ideal Tolerante a Falhas

### 3.1.4 Ações Atômicas

O conceito de ações atômicas originou-se de pesquisas em gerência de bancos de dados e em conseqüência desse fato, ficou inicialmente restrito a essa área. Um dos primeiros experimentos que estenderam o uso de ações atômicas surgiu em [39]. Em [30] encontramos também outro conceito de ações atômicas em relação aos mecanismos de tolerância a falhas.

A atividade de um componente é definida ser a seqüência de transições do estado externo de um componente. Cada transição é considerada como primitiva e indivisível. Uma transição única de estado constitui a mais simples ação atômica. Claramente, não pode haver interações entre o componente e o resto do sistema durante o período de uma transação única. A ausência de interações pode ser tomado como um critério para **atomicidade**:

⇒ A atividade de um grupo de componentes constitui uma ação atômica se não há interações entre aquele grupo e o resto do sistema durante a atividade (do grupo).

Para o resto do sistema, tudo o que existe da atividade dentro de uma ação atômica parece ser indivisível e pode ocorrer instantaneamente a qualquer hora durante a ação atômica. A Figura 3.3 mostra a atividade de três componentes particionados dentro de um número de ações atômicas indicado por linhas curvas, com linhas verticais usadas para mostrar as interações entre aqueles componentes. Inicialmente os componentes 1 e 2 interagem na ação atômica A, enquanto a atividade do componente 3 sozinho formou uma ação atômica privada. Depois de um pequeno período de tempo durante o qual todos os três componentes puderam se interagir, o componente 1 entrou em uma ação atômica privada, enquanto os componentes 2 e 3 entraram na ação atômica B. Consequentemente, os componentes 2 e 3 entraram em ações atômicas privadas aninhadas dentro de B. Note que dentro de cada ação atômica pode haver muitas ações atômicas aninhadas como foi mostrado; onde, em cada caso apenas a ação atômica com um período mais longo foi mostrada.

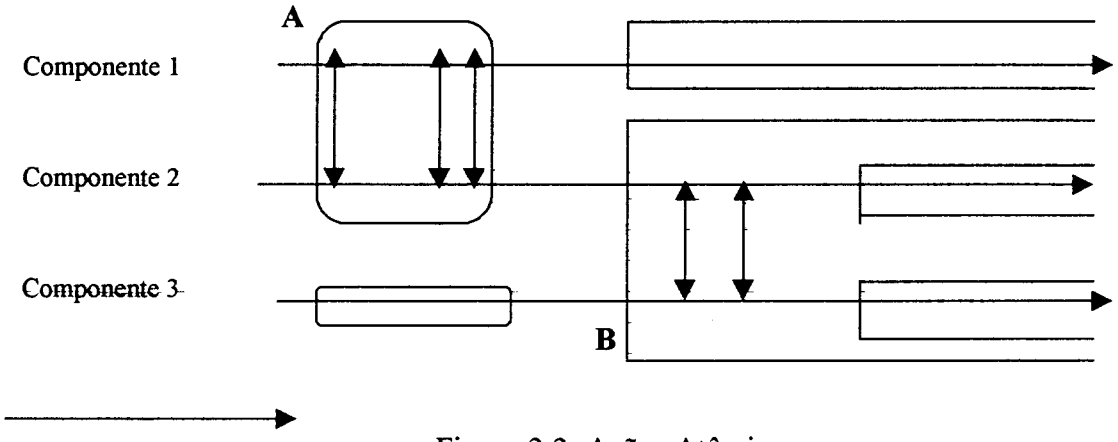



Figura 3.3 Ações Atômicas

De acordo com Randel et al [47] há equivalentes modos de se expressar as propriedades de ações atômicas:

- Uma ação é atômica se os processos que a estão realizando não estão conscientes da existência de qualquer outro processo ativo, e nenhum outro processo ativo tem consciência da atividade dos processos durante o tempo em que os processos estão realizando a ação atômica;

- Uma ação é atômica se os processos realizando-a não se comunicam com outros processos enquanto a ação está sendo realizada;

- Uma ação é atômica se os processos realizando-a não podem detectar nenhuma mudança de estado exceto aquelas mudanças realizadas por eles mesmos e esses estados não são revelados até que a ação atômica termine;

- Ações são atômicas se elas podem ser consideradas serem indivisíveis e instantâneas, assim como os processos estão preocupados em ser, de forma que os efeitos sobre o sistema são serializados para garantir a concorrência.

Ações atômicas provêm ao projetista de um sistema uma ferramenta conceitual útil para fabricar e organizar as interações entre os componentes do sistema. Essa ferramenta é particularmente efetiva quando ações atômicas podem ser forçadas a impor restrições sobre as interações dos componentes, além de impor estrutura sobre o fluxo de informação dentro de um sistema. Técnicas para forçar a atomicidade aplicam o uso de separação física (por exemplo, o uso de processos separados) para controlar as facilidades para interação (por exemplo, observando objetos compartilhados). Como tem sido visto, o fluxo de informação (particularmente de informação errônea) é uma importante consideração para tolerância a falhas que faz uso do conceito de atomicidade, principalmente quando os mecanismos de tolerância a falhas são aplicados em sistemas concorrentes e distribuídos.

### 3.1.5 Recuperação de Erros

Quando uma falha for capaz de gerar erros no estado de um sistema, o projetista do sistema deve crer que uma previsão efetiva do dano total pode ser feita, ou que o dano pode ser efetivamente avaliado. Quando esse é o caso, o dano previsto é dito ser um dano antecipado. Quando uma previsão efetiva ou avaliação não pode ser feita, o dano do sistema é dito não-previsto. A discussão entre danos que são previstos (antecipados) e os que não são leva-nos a duas técnicas totalmente diferentes para recuperação de erros: **recuperação de erros com avanço** e **recuperação de erros com retrocesso**. Para a situação onde o dano é previsto, a técnica para recuperação de erros é implementada de maneira eficiente, tornando específicas as mudanças limitadas (as mudanças são limitadas normalmente às áreas identificadas do dano e aplicadas aos erros específicos) para o estado do sistema. Em outras palavras, o estado errôneo atual é manipulado pelo sistema em ordem para remover os erros e habilitar o sistema a mover-se adiante sem falhar. Isso refere-se diretamente à técnica de recuperação de erros com avanço[51,57].

As características de uma técnica para recuperação de erros com avanço são:

⇒ dependente da avaliação do erro;

⇒ apropriada a sistemas particulares (aqueles cujos erros são previstos);

⇒ imprópria para dar suporte à recuperação de erros não previstos pelo sistema;

⇒ apropriada para aplicações com sistemas de tempo-real e sistemas críticos.

Complementar à técnica de recuperação de erros com avanço, a técnica de recuperação de erros com retrocesso retorna o sistema a um estado anterior ao erro, ou seja, livre de erro. A recuperação de erros com retrocesso[5,6,27,38,57] pode ser provida como uma medida ou um mecanismo em uma variedade de modos, cada uma oferecendo diferentes medidas em termos de custos em tempo de execução e armazenagem, contra os benefícios de flexibilidade e segurança. Alguns desses benefícios se baseiam na principal meta da técnica de recuperação: restaurar o sistema a um estado anterior, na esperança de que esse estado anterior esteja livre de erros. Essa técnica de recuperação segue às seguintes características:

⇒ independe da avaliação do dano;

⇒ capaz de prover recuperação às falhas arbitrárias;

⇒ é um conceito geral aplicável a todos os sistemas e;

⇒ afeta a velocidade do sistema.

A independência da avaliação do dano é uma conseqüência direta através da premissa de que qualquer parte do estado do sistema pode conter um erro. A recuperação de erros com retrocesso deve, no entanto, atingir uma completa troca de estados incorretos por outros corretos (ou supostamente livres de erros), onde, nesse caso, fica desnecessário a ação de avaliação do dano. Por essas características e por ter a habilidade de realizar correta ação de recuperação sem se preocupar com qualquer suposição sobre a falha, a recuperação de erros com retrocesso é uma técnica extremamente poderosa e muito mais utilizada, pelos motivos apresentados, que a técnica de recuperação de erros com avanço.

A restauração de estado é mais apropriada e pode, no entanto ser mais cara para se implementar, mas ela tem a enorme vantagem por conseguir recuperar-se de erros gerados por quaisquer falhas, incluindo os danos não previstos pelas falhas de projeto. Uma outra

vantagem é que este enfoque pode ser considerado como um mecanismo, que se divide em três categorias:

- *mecanismos de "checkpoint"* [34]: que salvam uma cópia de todos (algumas vezes parte) do estado do sistema;
- *técnicas "audi trail"*: que registram todas as modificações feitas para o estado;
- *mecanismos de "recovery cache"*[28]: um meio termo que registra uma cópia de somente aquela parte do estado que foi mudado.

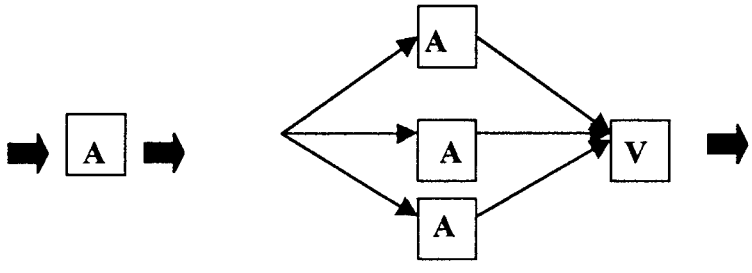
A geração destes mecanismos em um sistema de processo único é relativamente simples, mas é mais difícil para sistemas hierárquicos e aqueles que contêm processos concorrentes. Esta técnica é obviamente vantajosa pois ela independe da avaliação do erro e é capaz de fornecer recuperação a erros arbitrários, mas mesmo assim, essas vantagens acabam por gerar algumas desvantagens, como por exemplo: a velocidade de operação do sistema é afetada pelo fato da restauração ao estado anterior à falha, e também a necessidade de dispositivos extras para armazenamento de informações, o que torna o sistema mais caro. Exemplos de como ambas as técnicas de recuperação de erros, com avanço e com retrocesso, são implementadas usando a notação C++, são mostrados em [11] e [51].

### 3.1.6 Redundância de Hardware

A redundância de hardware pode ser definida para criar tolerância em componentes físicos de hardware, como portas lógicas, unidades de memória, transistores e etc.

Há duas categorias de redundância de hardware: redundância de hardware estática e redundância de hardware dinâmica. Na redundância de hardware estática, os componentes redundantes são usados dentro de um sistema para proporcionar tolerância a falhas de tal modo que os efeitos de um fracasso no componente sejam escondidos e não percebidos

pelo sistema. Um exemplo canônico de redundância estática é o TMR[5] que provê tolerância diante da replicação de módulos de hardware. Na Figura 3.4a, o módulo A poderia ser trocado pelo esquema da Figura 3.4b, onde há cópias idênticas do módulo A e um módulo V que seleciona a melhor e mais confiável saída. Já a redundância dinâmica proporciona uma capacidade de detecção de erro dentro de um sistema, e tem que ser suprida pela redundância em outro lugar para conseguir alcançar a tolerância a falhas. Por exemplo, o uso de códigos de correção de erros em unidades de memória podem ser considerados com o uso de redundância estática, no entanto, um simples código de paridade pode ser tratado com o uso de redundância dinâmica, já que tolerância de uma falha de memória deverá requerer redundância em outro lugar, por instante, no código do sistema operacional.




**3.4.a** **3.4.b**

Figura 3.4 TMR -Triple Modular Redundancy

[5] Do inglês: Triple Modular Redundancy

### 3.1.7 Redundância de Software

Tolerância a falhas de software requer redundância de projeto assumindo que a independência relativa dos esforços de programação irão reduzir a probabilidade de ocorrerem falhas de software idêntico em mais de uma versão do programa. Portanto, tolerância a falhas de software requer uma redundância não somente como uma replicação de programas como em hardware, mas também uma redundância de projeto.

As redundâncias de software também são divididas em duas categorias: redundância de software estática e redundância de software dinâmica[45]. A redundância de software estática usa componentes de software extras dentro de um sistema cujos efeitos de um ou mais erros podem ser escondidos e não percebidos pelo sistema. Um exemplo de tolerância a falhas através de redundância estática é utilizar o esquema de programação de N-versões[2,5]. Este esquema baseia-se no fato de que N-versões de um programa são independentemente projetadas e executadas em paralelo; e seus resultados são comparados por alguma forma de votação [51].

A redundância dinâmica de software consiste de diversos componentes redundantes onde somente um componente é ativo por vez, ou seja, sua execução não é em paralelo como a redundância estática. Se um erro no software é detectado em um componente ativo, então este é trocado pelo próximo componente disponível. Um exemplo de redundância dinâmica é bloco de recuperação[5,46]. Neste contexto o que acontece é que somente a primeira variante (chamada de alternante primário) opera e um teste de aceitação é aplicado com o propósito de detectar um erro; se o resultado não passar no teste de aceitação, então o estado do sistema é restaurado e a segunda variante é executada com os mesmos dados de entrada, e assim seqüencialmente, até que o resultado passe no teste ou todas as variantes sejam esgotadas (ver Seção 3.2.2 para maiores detalhes).

## 3.2 Tolerância a falhas de software

Esta seção discute um mecanismo de recuperação de erros e duas técnicas para a implementação de tolerância a falhas de software que são: **Tratamento de Exceções, Bloco de recuperação** e **N-versões**, respectivamente. Em Purtilo e Jalote[44] encontramos a descrição de um sistema que suporta tanto o esquema de bloco de recuperação quanto o método de programação de n-versões.

### 3.2.1 Exceções e seus Tratamentos

Exceções[16,25,32,54] são eventos extraordinários detectados durante a execução de um programa. Esses eventos podem ser erros de programação ou também erros que são definidos pelo programador da aplicação. Em muitos casos, entretanto, elas podem ser causadas por eventos inesperados, mas supostamente corretos. Como conseqüência da geração de uma exceção, um programa pode ser levado a terminar precipitadamente, ou de outra forma, o programa pode continuar, mas gerar resultados errôneos.

Exceções detectadas a nível de hardware aparecem como "exceções de hardware" ou interrupções. Um exemplo seria a execução de uma instrução ilegal ou tentativa de divisão de um número qualquer por zero. Uma exceção detectada a nível do sistema operacional pode aparecer como um sinal, como o uso incorreto da memória.

Um sistema de tratamento de exceções especifica o comportamento do programa depois que uma exceção foi detectada. Esta especificação é dividida em duas partes: a primeira sendo para suprir instruções requeridas como uma conseqüência de uma exceção; e a segunda sendo para manter o fluxo de controle depois que as instruções tenham sido completadas. Conceitualmente, quando consideramos programas que utilizam o tratamento de exceções, dois conjuntos de códigos e dois modos de execução podem ser identificados. O código do programa pode ser separado em duas partes distintas: a parte com código para a execução normal e a parte com código para execução anormal, ou seja, para tratamento de

exceções. Essa separação leva-nos à definição do componente ideal, já apresentada na Seção 3.1.3.

Uma execução excepcional ou anormal é aquela em que ocorre depois que uma exceção foi detectada, sendo esta tratada pelo **tratador de exceção**.

Depois que uma tratador de exceção é executado, o controle de um programa pode ou ser transferido para algum lugar fora do código do tratador, ou a execução do programa pode simplesmente acabar. Esta discussão, segundo [10,23], leva-nos a classificar os modelos de tratamento de exceções em duas categorias:

- **modelo de terminação**[6]*:* neste modelo, quando um programa chama um procedimento e este gera uma exceção, o controle retorna imediatamente ao local do programa onde o procedimento que gerou a exceção foi chamado;
- **modelo de retomada ou continuação**[7]*:* mesma idéia que o modelo de terminação, a única diferença é que o controle ao invés de retornar para o programa que chamou o procedimento, ele simplesmente retorna para o ponto seguinte onde a exceção foi gerada.

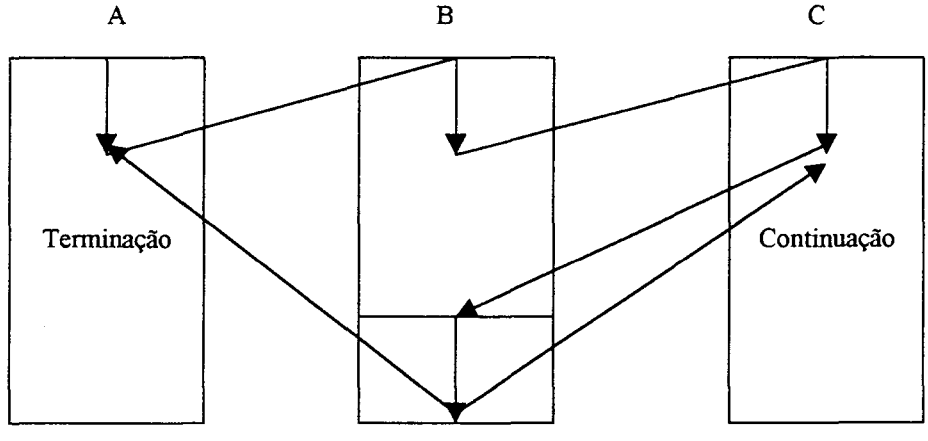



Figura 3.5: Modelos adotados pelos mecanismos de tratamento de exceções

[6]Do inglês : termination model
[7]Do inglês : resumption model

A Figura 3.5 é composta de três procedimentos: A, B e C. O procedimento A invoca o procedimento B e este por sua vez invoca o procedimento C. Durante a execução do procedimento C, uma exceção é gerada, sendo que essa exceção é tratada pelo procedimento chamador, ou seja, o procedimento B, já que não existe nenhum tratador local. A execução de C é então suspensa e o controle é transferido para o tratador associado. Se o modelo de retomada é adotado, então a execução de C é retomada imediatamente após o ponto em que a exceção foi detectada. Se o modelo de terminação foi adotado, então a execução continuará logo após a chamada do procedimento que gerou a exceção, ou seja, o controle é retomado pelo procedimento A.

Argumentos convincentes de que o paradigma de terminação é superior ao paradigma de retomada são apresentados em [58]. Yemini e Berry[63] identificaram que no modelo de terminação é possível obter três respostas distintas: reexecução de quem sinalizou a exceção, propagação da exceção ou seu possível retorno. Já no modelo de retomada, o tratador da exceção tem a capacidade de retornar à atividade do componente ao ponto seguinte em que a exceção foi sinalizada. Experimentos com o uso do mecanismo de retomada indicam que este tipo de mecanismo pode ser propenso a falhas; por isso o mecanismo de terminação é usado pela maioria das linguagens de programação, como por exemplo, Ada[54] e C++[58].

O tratamento de exceções é um mecanismo para recuperação de erros de software baseado na recuperação de erros com avanço e que pode ser usado para estabelecer uma infra-estrutura de apoio a sistemas tolerantes a falhas de software.

Três aspectos devem ser levados em consideração para a implementação dos mecanismos de tratamento de exceção: a representação da exceção, a associação entre tratadores e exceções e o local de definição dos tratadores. Esses três aspectos serão discutidos a seguir.

### Representação das exceções

A representação da exceção refere-se a um tipo de representação interna que diferencia uma exceção em particular das demais exceções. Ela pode ser classificada nas seguintes categorias:

- **Exceções como Cadeias de Caracteres**: exceções são implementadas como cadeias de caracteres. Este enfoque é o mais tradicional adotado pela maioria das linguagens imperativas, como CLU[40] e Ada[54] e linguagens orientadas a objetos como Módula-3[52], Guide[4,35,36], Eiffel[43] também utilizam este enfoque;

- **Exceções como objetos de dados**: exceções são classes[9] e, toda vez que uma exceção é gerada, uma instância dessa classe é criada. Desse modo, o tratador irá receber como parâmetro o objeto (que corresponde a exceção). Linguagens como C++ e Arche[7] utilizam este enfoque.

- **Exceções como objetos completos**: Exceções são hierarquicamente organizadas em classes baseadas em comportamentos em comum: a exceção "divisão-por-zero" é naturalmente uma subclasse da classe exceções-aritméticas. Quando uma exceção é gerada, uma instância de uma classe é criada. A diferença entre essas exceções e as que são representadas por objetos de dados, é que as exceções representadas como objetos completos possuem comportamento, e as exceções representadas como objetos de dados são somente repositórios de dados. Linguagem como Lore[5,6] utiliza esse enfoque.

### Associação entre exceções e tratadores

Dois tipos de associação entre exceções e tratadores e são criadas quando uma exceção é gerada, que são: **associação estática** e **associação dinâmica**.

Quando uma associação é estática, significa que ela ocorreu durante o tempo de compilação. Nesse tipo de associação, o código do tratador é associado com uma região de

código do programa, chamada de região restrita ou protegida (Figura 3.6b), onde a execução normal acontece. Com isso, fica nítida a separação entre a execução de código normal e a execução de código anormal. A maioria das linguagens de programação utilizam associação estática.

A associação dinâmica depende do fluxo de execução do programa, ou seja, dependendo de onde a exceção foi gerada e como é o fluxo de execução, ela pode ser tratada por diferentes tratadores (Figura 3.6a). A linguagem de programação PL/I utiliza este tipo de associação.

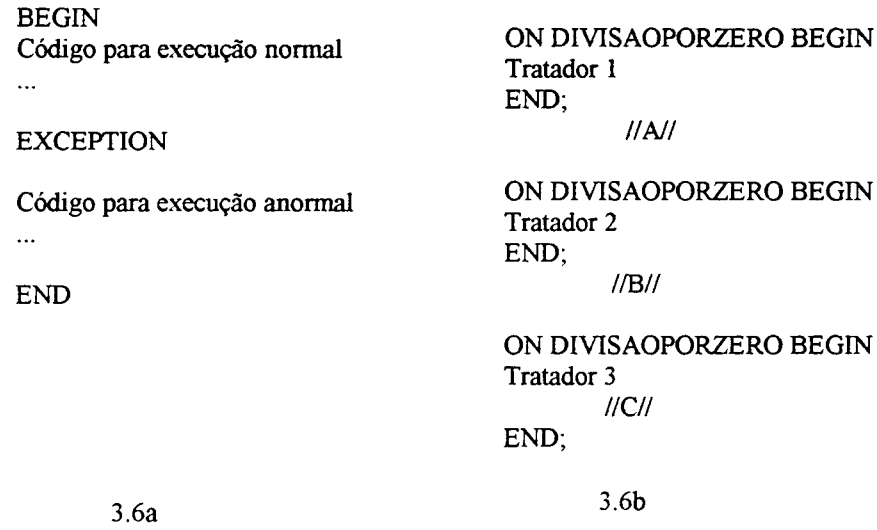



Figura 3.6: Exemplos de Associação estática e dinâmica

## Localização dos Tratadores

O local dos tratadores indica os diferentes modos que um programador pode escolher para ligar uma exceção a um tratador. O local de associação pode ser: um comando, um bloco de comandos, um objeto ou uma classe.

A localização dos tratadores associada a um bloco de comandos é adotada pela maioria das linguagens de programação, como por exemplo, C++ (Figura 3.7). A linguagem de programação CLU utiliza a associação de tratadores a um comando. Já a associação de tratadores a classes definem comportamentos comuns a todas as instâncias de uma classe em situações excepcionais. Linguagens como Lore e Guide utilizam esse

enfoque. Cui e Gannon[17] descrevem uma implementação na linguagem de programação Ada para este tipo de associação.

```
Enum matherr { Overflow, Underflow, ZeroDivide}

Try{
 // ...
 if (...) THROW Overflow;
}

Catch (matherr m) {
 Switch (m) {
  Case Overflow:
       // .....
  Case Underflow:
       // ...
 }
//...
```

Figura 3.7: Exemplo de Tratamento de exceções em C++

Tratadores de exceções em C++ estão associados a blocos de comandos. Os tratadores são declarados no final do bloco TRY através da cláusula CATCH. Uma exceção pode ser diretamente sinalizada em um bloco TRY usando a expressão THROW.

Exceções em C++ podem ser declaradas como classes e uma instância de uma dessas classes é criada toda vez que uma exceção for sinalizada. Sinalizar uma exceção é passar um objeto exceção como parâmetro para um tratador.

### 3.2.2 Bloco de Recuperação

O bloco de recuperação[46] é uma técnica aplicada a sistemas seqüenciais e que utiliza redundância dinâmica. Neste esquema, as versões são nomeadas de variantes e o teste de aceitação é chamado de seletor.

Na entrada do bloco de recuperação, o estado do sistema deve ser salvo para que se possa permitir a recuperação de erros com retrocesso, ou seja, para estabelecer um *checkpoint* [46]. A sintaxe do Bloco de Recuperação é mostrada na Figura 3.8.

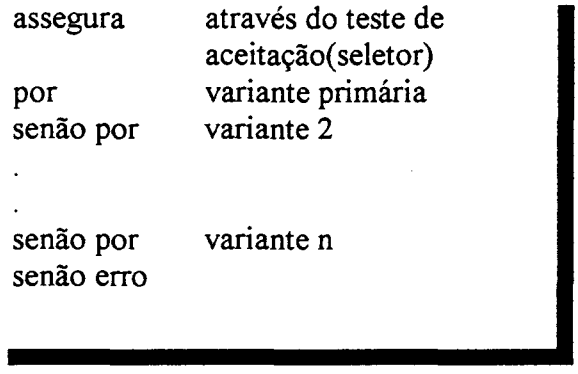



Figura 3.8: Sintaxe do bloco de recuperação

A variante primária é executada e então o teste de aceitação é aplicado aos resultados obtidos pelas variantes. Se o teste de aceitação ocorre com sucesso, então é feita uma saída no bloco de recuperação e é descartada a informação guardada para se efetuar o *checkpoint*. No entanto, se o teste de aceitação não ocorre com sucesso, ou seja, o resultado não é aceito, ou se ocorrem problemas durante a execução do módulo, então uma exceção é sinalizada e a recuperação de erros com retrocesso é invocada. Isto retorna o sistema ao estado salvo anteriormente na entrada. Depois disso, a próxima variante é executada e o teste de aceitação é aplicado novamente e assim até que todas as variantes se esgotem. Se todas as variantes se esgotaram e nenhuma passou no teste de aceitação, então uma exceção de falha irá ser transmitida para o ambiente do bloco de recuperação.

O teste de aceitação é um módulo que deve ter tolerância a falhas a fim de não falhar na hora de decidir sobre os resultados obtidos das variantes. Este tipo de tratamento da falha é o escopo da técnica de recuperação de erros com retrocesso que tratam exatamente da restauração de estados do sistema.

### 3.2.3 N-Versões

O esquema de N-versões[2] é uma técnica de tolerância a falhas de software que utiliza redundância estática. N-versões de um programa são independentemente projetadas para satisfazer uma especificação em comum e são executadas em paralelo, e seus resultados são comparados através de um mecanismo de votação que determina um único resultado correto. Baseado no voto majoritário, esse mecanismo de votação, que nada mais é que um seletor[8] pode eliminar resultados errôneos (ou seja, eliminar a minoria) e passar os resultados (assumidos estarem corretos) para o resto do sistema. A Figura 3.9 ilustra essa técnica. Esta técnica é análoga à técnica de tolerância a falhas de hardware (ver seção 3.1.6 – Redundância de hardware), a TMR, onde a execução em paralelo é assumida. Cada um dos esquemas listados acima, tanto o bloco de recuperação quanto N-versões podem ser convenientes para uma classe de aplicações e os seus potenciais para melhorar a confiabilidade em software têm sido demonstrados por vários experimentos.

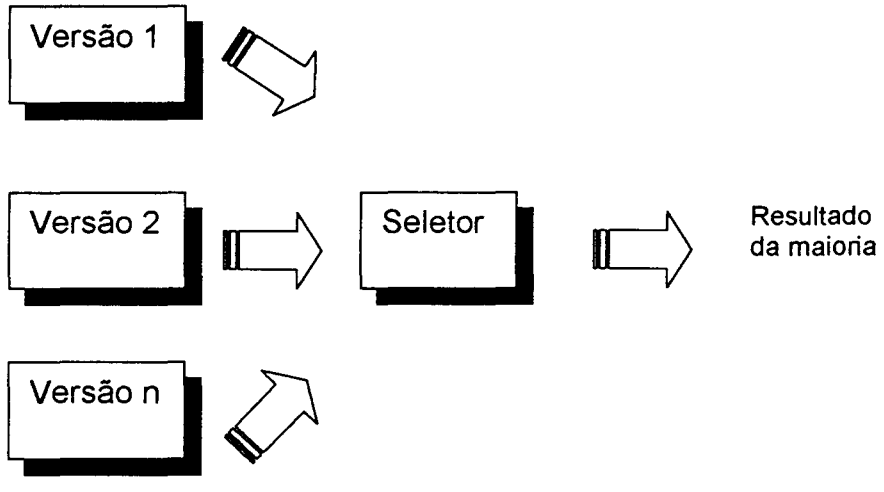



Figura 3.9: N-versões

[8] Do inglês: voter

## 3.3 Tolerância a Falhas de hardware orientada a objetos

O uso do modelo orientado a objetos para o projeto de sistemas pode reduzir a complexidade do sistema permitindo a decomposição de um software em um conjunto de componentes claramente separados, de forma que os mesmos componentes possam ser usados na construção de diversas aplicações relacionadas, ou seja, eles podem ser reutilizados. Em relação à tolerância a falhas de hardware, técnicas orientadas a objetos usando componentes reutilizáveis podem ser também empregadas.

Um exemplo disso é o enfoque utilizado pelo sistema Arjuna[22,55]. Arjuna provê um conjunto de ferramentas que auxiliam a construção de aplicações distribuídas tolerantes a falhas utilizando o conceito de ações atômicas (ver Seção 3.1.4) que operam sobre objetos persistentes. Toda funcionalidade do Arjuna é fornecida através de classes, ou seja, além de dar suporte ao modelo de objetos, toda a sua estrutura interna também é orientada a objetos, isto permite ao usuário definir objetos por meio do mecanismo de herança e incorporá-los as suas aplicações.

### 3.3.1 Um ambiente para Programação distribuída: Arjuna

A implementação do framework proposto nessa dissertação é baseada também no sistema Arjuna. Arjuna é um ambiente implementado em C++. Em Arjuna os objetos são atômicos e persistentes. As ações atômicas controlam as operações sobre os objetos de forma a garantir que somente as mudanças consistentes de estados ocorram apesar de falhas e de acessos concorrentes.

Os componentes de hardware em Arjuna são considerados como estações de trabalho conectadas por um subsistema de comunicação. Essas estações operam normalmente ou simplesmente param de trabalhar. Após uma parada, a estação é reparada em um tempo limitado e reintegrada ao sistema. Com isso, supõe-se que todo o dado não armazenado em memória estável é perdido durante a parada; o dado armazenado em

memória estável é inalterado pela parada eventual de uma estação. Os protocolos de comunicação implementados pelo módulo RPC[9], (Figura 3.10) toleram duplicação, corrupção e perda de mensagens.

A arquitetura do sistema Arjuna é composta dos seguintes módulos: **RPC, Name Server, Atomic Action** e **Object Store**. A Figura abaixo mostra como esses módulos se interagem:

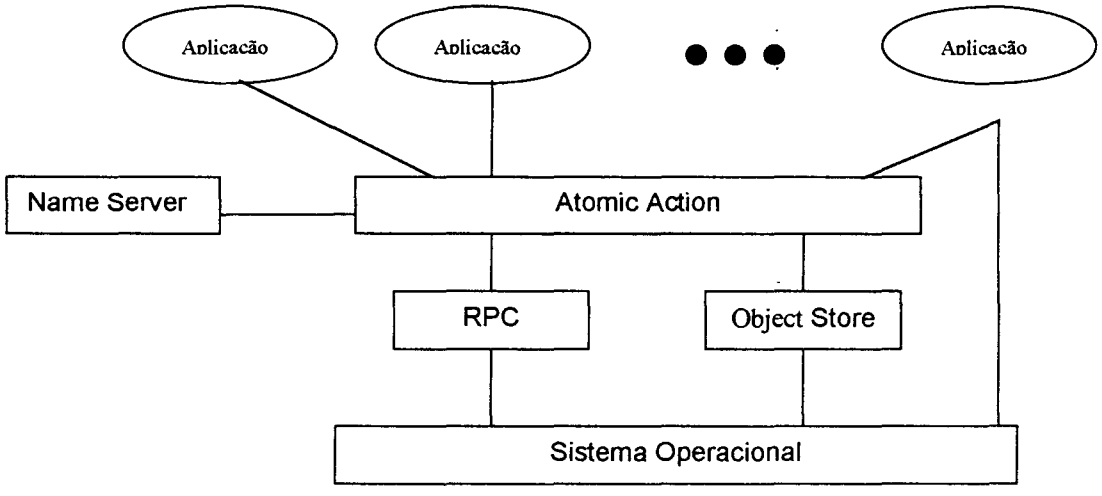



Figura 3.10: Arquitetura do sistema Arjuna

Em Arjuna, os objetos persistentes de uma aplicação distribuída são estruturados através do modelo cliente-servidor. Um servidor gerencia o estado de um objeto e define e executa operações que são exportadas aos clientes. Já os clientes ativam essas operações no intuito de alterar o estado do objeto controlado pelo servidor. O módulo RPC se encarrega de executar todas essas ativações de operações que são implementadas através de chamadas de procedimento remoto. O módulo Name Server gerencia os nomes que são utilizados para referenciar objetos. Função de gerência de resolução de nomes são necessárias para a implementação de sistemas computacionais em geral, pois a função mapeia o identificador

[9] Do inglês: Remote Procedure Call(RPC)

único para tornar viável a sua localização em um sistema distribuído. O módulo Atomic Action fornece a interface de programação do Arjuna. Possui mecanismos necessários para a implementação de controle de concorrência, persistência, recuperação de estados e controle de ações atômicas. A estrutura interna de Atomic Action é composta de uma série de classes, dentre elas a classe Atomic Action. Para estruturar um sistema como um conjunto de objetos atômicos, o programador declara instâncias da classe Atomic Action promovendo operações como : `begin()`, `end()` e `abort()`, mostrado na Figura 3.11.

```
Object object; // Cria uma instância de um objeto
...
AtomicAction A; // cria uma instância da classe Atomic Action
...
A.Begin() // Inicia uma ação atômica
...
Object.Op(); // Executa alguma operação no objeto
...
A.End(); // Confirma a ação atômica
```

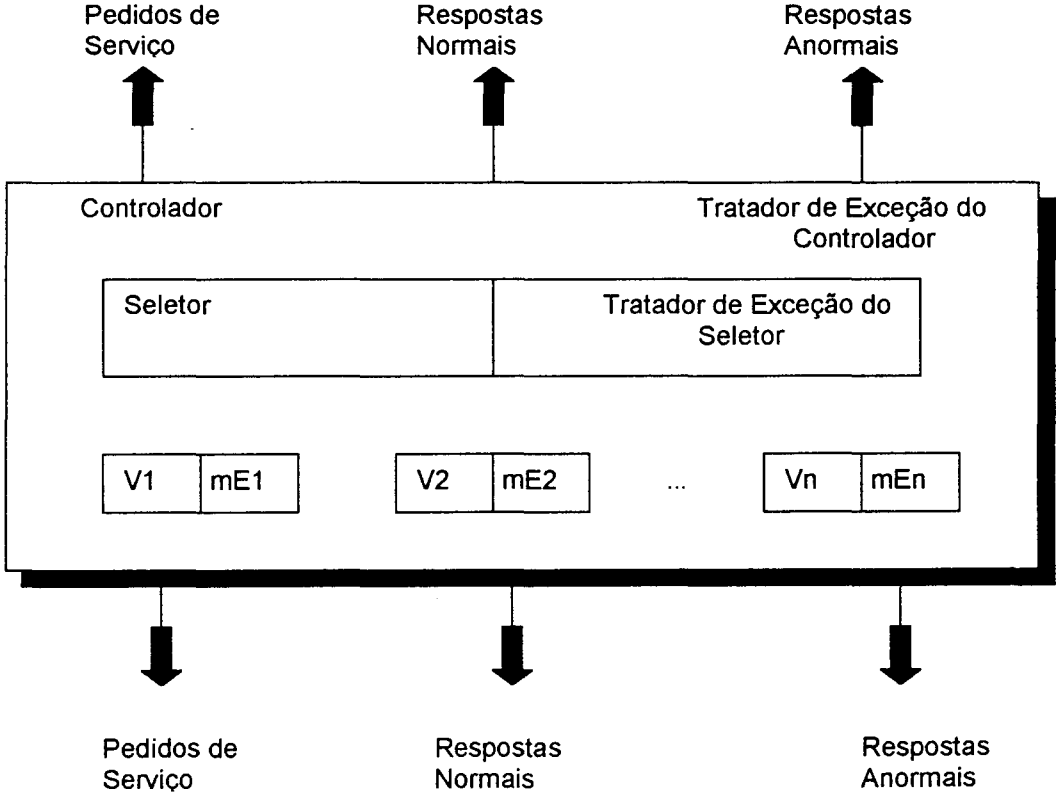
Figura 3.11: Objetos de Ações Atômicas

O módulo Object Store fornece o acesso a um serviço de gerência de estado persistente de objetos. O repositório de objetos armazena o estado passivo de objetos persistentes em memória estável. A representação do estado passivo dos objetos é independente de arquiteturas de máquinas, permitindo que estados de objetos sejam transportados sem problemas entre as memórias estável e volátil, bem como em mensagens de comunicação entre máquinas.

A aplicação discutida nessa dissertação utiliza o ambiente Arjuna para dar suporte ao hardware, ou seja, para usar o mecanismo de tolerância a falhas de hardware que ela provê e também para fazer uso das ações atômicas, necessárias nesse caso para o mecanismo de bloco de recuperação afim de estabelecer a restauração de estados.

## 3.4 Tolerância a Falhas de software orientada a objetos

⇒ Em sistemas complexos, apesar do uso de técnicas de tolerância a falhas serem efetivamente empregadas, podem ainda restar falhas residuais. Diante desse fato, tolerância a falhas de projeto é requerida se há necessidade de tratar essas falhas, e sua provisão apenas pode ser alcançada se diversidade de projeto (Figura 3.12) for aplicada antecipadamente ao sistema. A ferramenta de controle (Controlador) corresponde à execução de uma das técnicas de tolerância a falhas de software: bloco de recuperação ou n-versões.




V1..Vn = módulos de diferentes projetos apontando para uma especificação em comum

mE1...mEn = tratadores de exceção para os módulos V1..Vn.

Figura 3.12: Estrutura da Diversidade de Projeto

Considerando uma abordagem orientada a objetos para implementar a redundância de projeto, podemos encontrar diferentes níveis de abstração que incorporam a redundância de software necessária. Estes níveis se dividem em:

⇒ Diversidade de Operação: as versões são implementações de operações da classe e desenvolvidas seguindo uma especificação em comum. As operações para prover tolerância a falhas de software podem ser operações privadas à classe;

⇒ Redundância de Objeto: tolerância a falhas é obtida através da instanciação de diversos objetos de uma mesma classe; onde esses objetos são distintos apenas em seus dados internos;

⇒ Diversidade de Classe: as versões são objetos de classes diferentes, onde dados e operações são implementados independentemente seguindo uma mesma especificação.

A abordagem mais utilizada é a Diversidade de classe, onde as especificações dos serviços são representadas por classes abstratas e as variantes são instâncias de subclasses concretas diferentes que realizam a especificação definida pela classe abstrata. Considere, por exemplo, uma classe abstrata `Pilha` e suas subclasses concretas `PilhaLista`, `PilhaVetor` e `PilhaDupla`, como mostra a Figura 3.13.

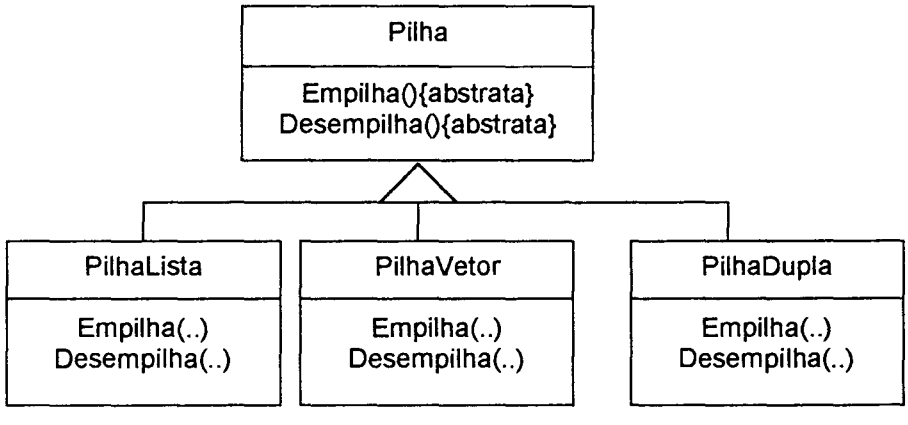



Figura 3.13: Exemplo de Diversidade de classes

Cada uma dessas subclasses representam variantes que executam operações que seguem uma especificação em comum, no caso, a especificação em comum corresponde à implementação de uma pilha utilizando-se as operações de `empilha()` e `desempilha()` através de uma lista ligada, um vetor e uma lista duplamente ligada. As operações `empilha()` e `desempilha()` são definidas na classe `Pilha` como sendo virtuais, sendo, portanto redefinidas pelas classes concretas. As variantes correspondem aos módulos de diferentes projetos (V1..Vn) e as técnicas de tolerância a falhas são utilizadas pela ferramenta de controle (Figura 3.12).

Embora essa abordagem seja cara devido à construção de diferentes variantes, ela permite que a construção de diferentes variantes seja feita através da reutilização de componentes, barateando o custo final do desenvolvimento do sistema. Uma outra vantagem do uso de diversidade de classe é que ela possibilita a construção de frameworks, já que a base de construção de frameworks fundamenta-se no conceito de reutilização sob a forma de classes abstratas que podem ser particularizadas para se adequarem às aplicações específicas.

O trabalho desenvolvido por essa dissertação é baseado na diversidade de classe construída através de um framework , onde as variantes são objetos de diferentes subclasses de uma classe base abstrata. O framework dá apoio à construção de aplicações tolerantes a falhas através de uma infra-estrutura composta por classes genéricas, provendo efetiva reutilização de componentes.

## 3.5 Sumário

Neste capítulo foram apresentados os conceitos fundamentais de tolerância a falhas. Tolerância a falhas é muito útil em sistemas onde a alta confiabilidade é essencial. A ausência da confiabilidade acarreta em erros até que às vezes irreparáveis para o sistema. Em geral, 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. Entretanto, a incorporação de redundância implica em custos maiores de

desenvolvimento do sistema. Além disso, a maneira com que ela é incorporada é relevante, pois ela pode aumentar ao invés de diminuir a complexidade.

Como foi mostrado no Capítulo 2, a programação orientada a objetos é considerada um modelo efetivo para a produção de software de melhor qualidade. Seus benefícios tais como: abstração de dados, encapsulamento, modularidade, polimorfismo e reutilização de componentes ajudam a implementar os mecanismos de tolerância a falhas facilitando o controle da complexidade do sistema, pois eles promovem uma melhor estruturação de seus componentes.

Foi também mostrado nesse capítulo que para as aplicações tolerantes a falhas distribuídas, além das falhas de software, as falhas de hardware devem ser consideradas, pois estamos tratando com ambientes de rede. O Arjuna provê suporte para falhas de hardware em um ambiente distribuído.

# CAPÍTULO 4

## FOOD : Um Framework Orientado a Objetos Distribuído para Tolerância a Falhas

O objetivo deste capítulo é a descrição do framework FOOD para tolerância a falhas de software. O framework apresentado tem as seguintes características:

1. Suporte de alto nível para tolerar falhas de software em uma camada de aplicação orientada a objetos;
2. Utilização do ambiente Arjuna para a provisão de restauração de estados através do uso de ações atômicas;
3. Uso da linguagem C++ para implementação das técnicas de bloco de recuperação e n-versões para prover tolerância a falhas de software;
4. Incorporação do mecanismo de tratamento de exceções em um ambiente distribuído.

O framework FOOD foi projetado de forma a utilizar integralmente os conceitos do paradigma de objetos, aproveitando-se de seus benefícios (como já mostrado no capítulo 2) criando assim uma estrutura modular de componentes genéricos que podem ser reutilizados para implementar os mecanismos de tolerância a falhas.

Esse capítulo tem a seguinte ordem: inicialmente na Seção 4.1 a descrição do framework orientado a objetos distribuído tolerante a falhas é mostrado; na Seção 4.2 um exemplo de uso do framework é mostrado através da implementação de uma Pilha Robusta; na Seção 4.3 são mostradas a validação e a avaliação do framework; e, finalmente, na Seção 4.4, o sumário deste capítulo é apresentado.

## 4.1 Descrição das classes do framework orientado a objetos distribuído - FOOD

A Figura 4.1 representa uma macro visão dos módulos que compõem o framework proposto. O modelo é representado por três módulos: o Cliente, o Network e o Arjuna, e suas dependências.

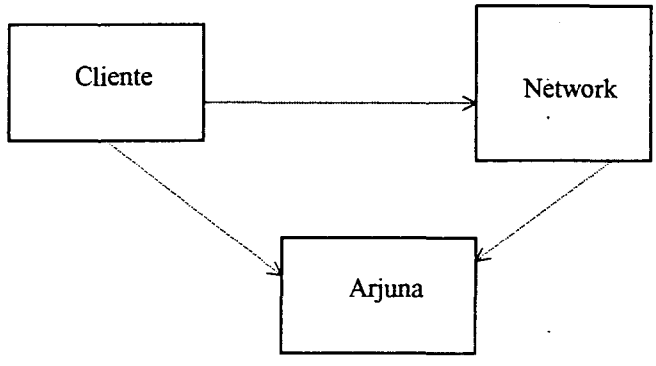



Figura 4.1: Módulos do framework e suas dependências

As classes e os relacionamentos do framework são detalhados na Figura 4.2. A seguir, descrevemos cada uma das classes do modelo, sendo que os detalhes de implementação podem ser encontrados no Apêndice A.

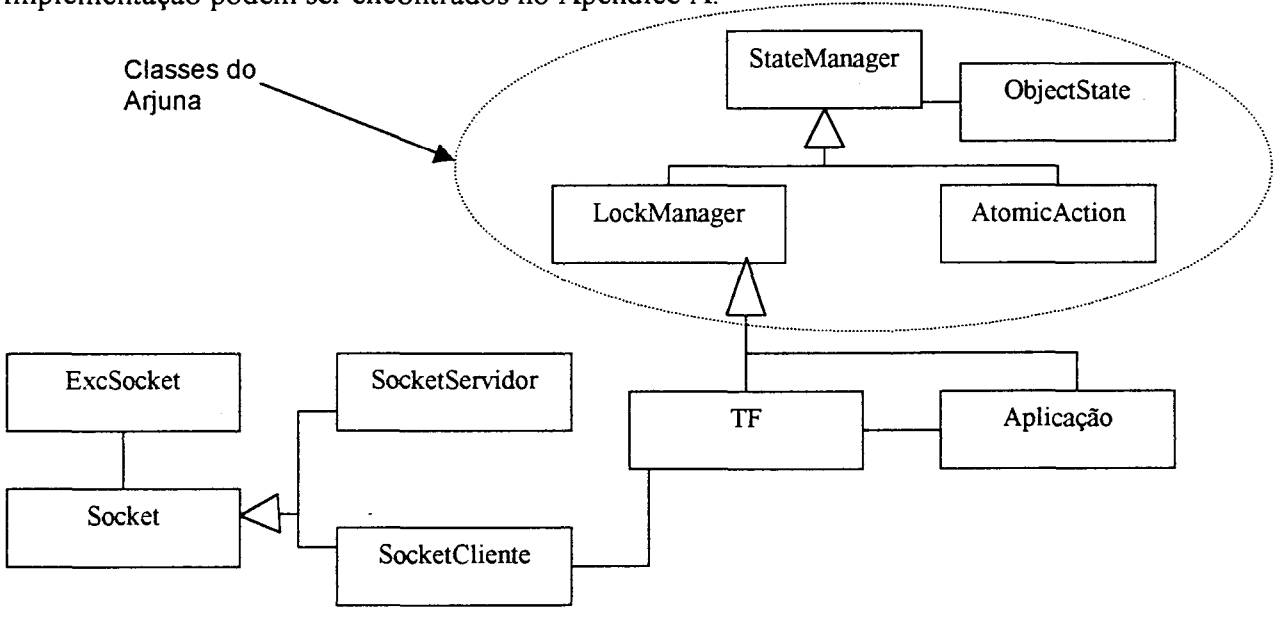



Figura 4.2: Classes e Relacionamentos do Framework

### 4.1.1 Classes do Arjuna

A classe `StateManager`, suas subclasses (`LockManager` e `AtomicAction`), e a classe `ObjectState` (Figura 4.3) , 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, respectivamente para criar, abortar e finalizar 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 ANDPERSISTEN) são recuperáveis e seu tempo de vida excede o tempo de execução da aplicação. Objetos do framework FOOD são criados automaticamente como ANDPERSISTENT.

A classe `LockManager`, por sua vez, usa as facilidades herdadas da classe `StateManager` e provê o 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.

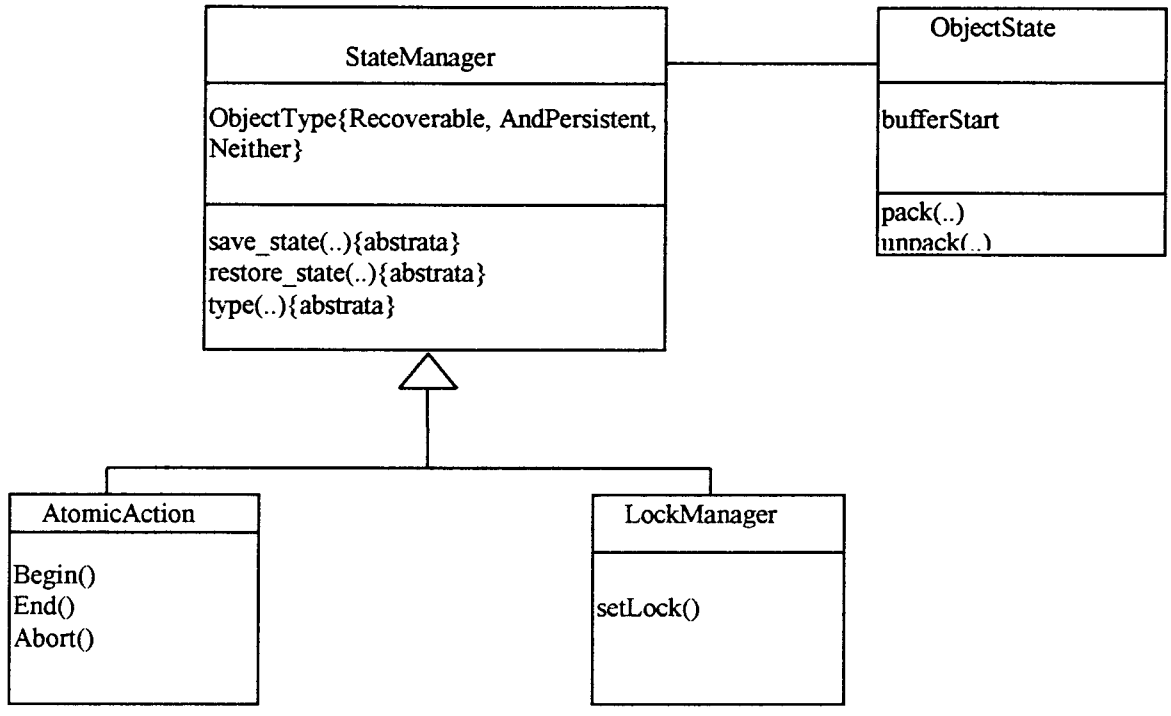



Figura 4.3 Classes do Arjuna

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

### 4.1.2 Classe ExcSocket

A classe `ExcSocket` é uma classe abstrata que pode ser especializada pela aplicação pelas próprias classes do framework para representar as exceções de Socket geradas durante a execução da aplicação, como por exemplo, exceções de criação, conexão, leitura, escrita, etc, como mostra a Figura 4.4.

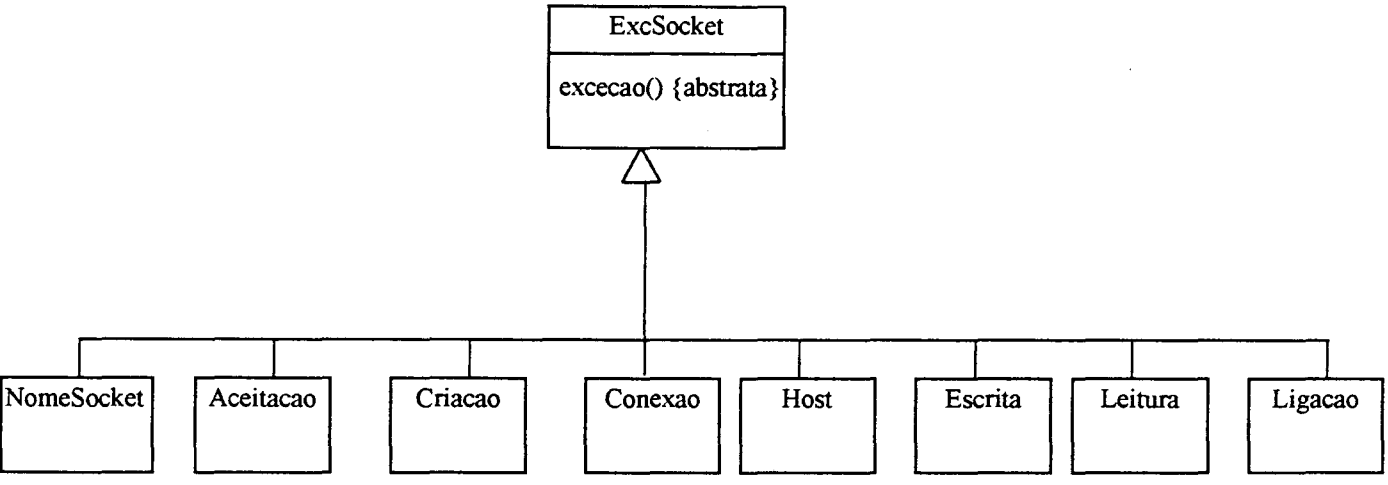



Figura 4.4: Classe `ExcSocket`

### 4.1.3 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()`.

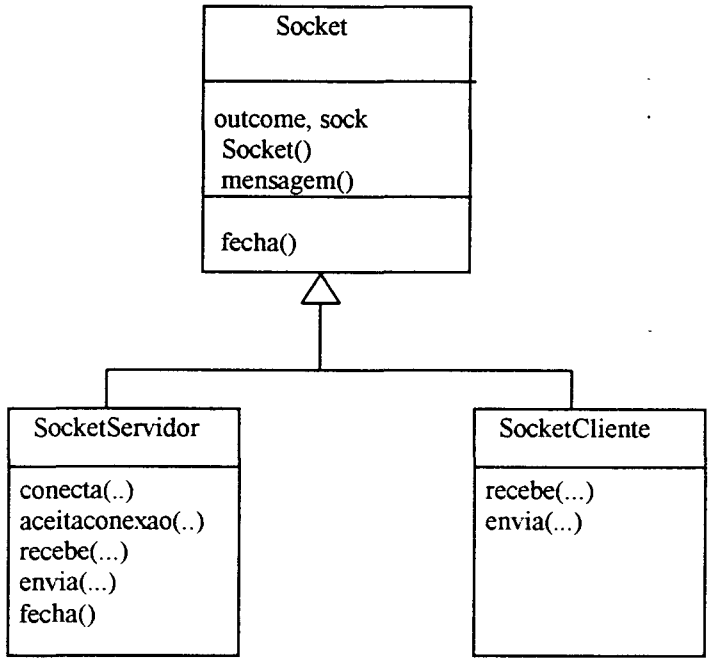



Figura 4.5: Classe `Socket`

### 4.1.4 Classe TF

Esta classe implementa as técnicas de tolerância a falhas de software: bloco de recuperação e n-versões. Ela é um tipo paramétrico (template) onde suas instâncias se comportam como classes tolerantes a falhas. Os atributos `port` e `maquina` guardam o port associado a cada processo servidor e a máquina onde eles executam. O atributo `obj` é uma referência para o objeto da aplicação. Ele é passado como argumento para os métodos

da classe que são tolerantes 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. 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. O método `Versao()` recebe como parâmetro dois arrays contendo os endereços das máquinas onde as versões executam e seus ports respectivamente. A definição da classe `TF` é apresentada abaixo.

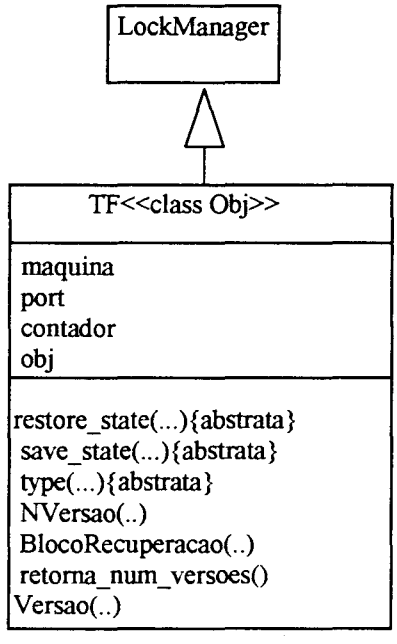



Figura 4.6: Classe `TF`

## Aspectos de Implementação da classe TF

`TF` é derivada de `LockManager`. Classes instanciadas a partir desta classe devem ser persistentes para permitir restauração de estados. Por isso, no momento de criação de

uma classe, o construtor de `LockManager` é invocado com o argumento ANDPERSISTENT. O código abaixo mostra a definição da classe `TF`:

```
Template<class Obj> class TF: public LockManager{
Int contador;
Int port[MAXVER];
Char* maquina[MAXVER];
Obj* obj;
virtual Boolean save_state(ObjectState& os, ObjectType ot) ;
virtual Boolean restore_state(ObjectState& os, ObjectType ot) ;
virtual  const TypeName type() const;
public:
 TF();
 ~TF() {LockManager::terminate();};
 Boolean NVersoes(Obj* obj);
 Boolean BlocoRecuperacao(Obj* obj1);
 Int retorna_num_versoes(){ return contador};
 Boolean  Versoes(int port, char* mach);
};
```

As implementações das técnicas de tolerância a falhas são mostradas abaixo.

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

```
1  template<class Obj> Bool TF::BlocoRecuperacao(Obj* obj1){
2  Socket_Server<Obj>* s[MAXVER];
3  obj = obj1;
4  for(int i=0;i<retorna_num_versao();i++){
5    s[i]=new Socket_Server<Obj>;
6  AtomicAction* A;
7  Int n = retorna_num_versoes();
8  For(int i=0; i<n; i++){
9      A = new AtomicAction;
10     A->Begin();
```

```
11    s[i]->envia(port[i],maquina[i],obj);
12    if(setlock(new Lock(write),0) ==GRANTED)
13      s[i]->recebe(obj)
14  else return (FALSE);
15  if(!obj->testeaceitacao())
16          A->Abort()
17   else{ A->Abort() ;
18        s[i+1]->Fecha();
19         delete[] s;
20       return TRUE;}
21  }
22  delete[] s;
23  A->End();
24  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 11). 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. Quando o resultado de uma variante é retornado (linha 13), o método `testeaceitacao()` é executado (linha 15). Caso o resultado seja aceitável, a transação é abortada e o valor `TRUE` é retornado; caso contrário, a transação é abortada e o estado anterior do objeto e passado para a próxima variante. Se todas as variantes forem executadas e nenhum resultado satisfatório for produzido, o valor `FALSE` é retornado e a transação é finalizada 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.

```
Template<class Obj> Bool TF::NVersoes(Obj* obj1){
  Socket_Server* s[MAXVER];
  Obj* 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;
```

## 4.2 Exemplo de Uso

O framework utiliza a linguagem de programação orientada a objetos C++ para o desenvolvimento da aplicação. O ambiente no qual foi desenvolvido o experimento descrito nessa seção consiste de um conjunto de estações rodando sob a plataforma UNIX, conectadas através do protocolo TCP/IP. Este modelo usa uma arquitetura cliente/servidor (Figura 4.7), onde clientes e servidores se comunicam através do uso de sockets.

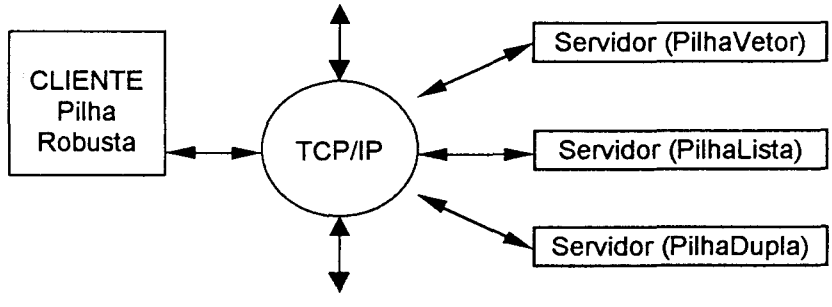



Figura 4.7: Arquitetura utilizada pela aplicação

As classes `PilhaLista`, `PilhaVetor` e `PilhaDupla` são classes concretas derivadas da classe `Pilha` e se comportam como suas variantes. O diagrama da Figura 4.2 foi refeito mostrando a Aplicação através da classe `Pilha` e suas versões.

A classe `PilhaRobusta`, também derivada da classe `Pilha`, utiliza os serviços de tolerância a falhas providos pela classe abstrata `Pilha`. Na Figura 4.8, as classes apresentadas são classes que pertencem ao framework e que são fornecidas pelo programador da aplicação.

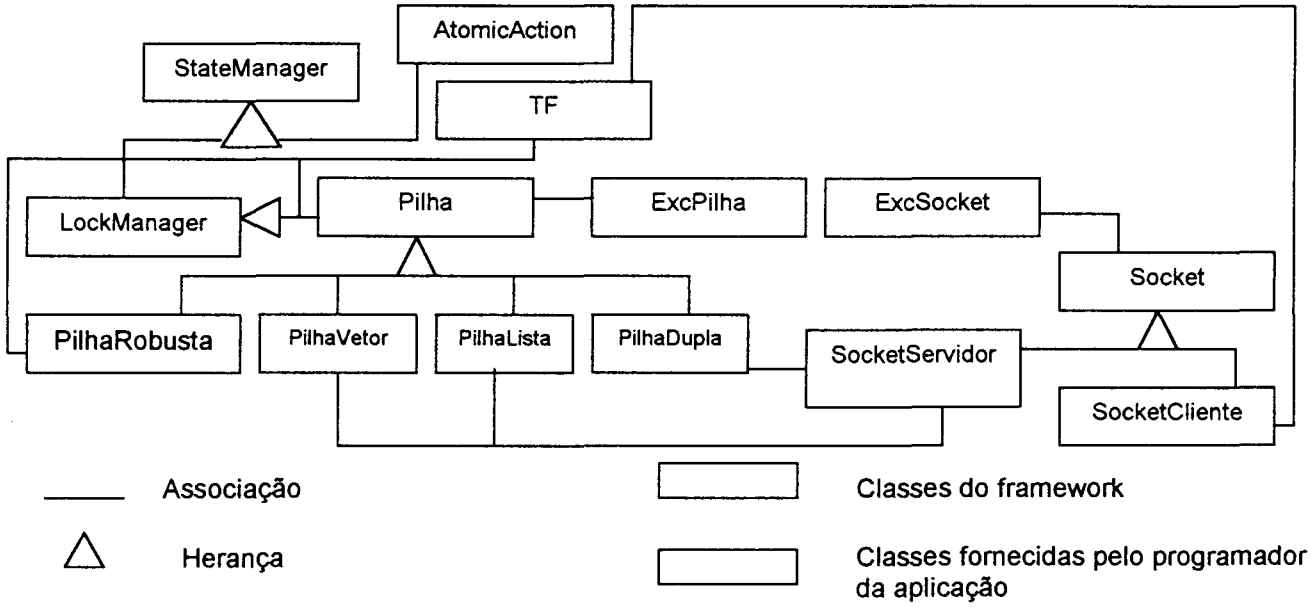



Figura 4.8: Relacionamento entre as classes do framework orientado a objetos distribuído para a aplicação da Pilha

### 4.2.1 Classe Pilha

`Pilha` (Figura 4.9) é uma classe abstrata que define a raiz da hierarquia para objetos tolerantes a falhas da aplicação, portanto, toda classe que quer ser tolerante a falhas da aplicação deve ser subclasse de `Pilha`. Os atributos protegidos da classe Pilha correspondem ao operador, ao item e ao resultado da exceção final, ou seja , se as variantes

executaram corretamente ou não e um tipo enumerado que retorna o tipo de exceção que pode ser gerada. A interface da classe `Pilha` é composta pelas operações virtuais `empilha()` e `desempilha()`, e pelas operações `seletor()`, `testeaceitacao()`, `marshalling()`, `unmarshalling()`, `save_state()`, `restore_state()` e `type()`, sendo que as três últimas operações são abstratas.

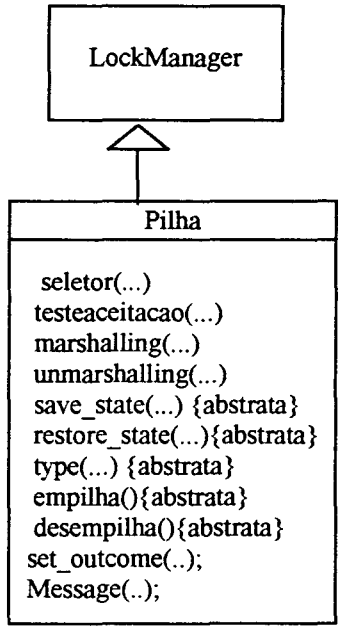



Figura 4.9 Classe `Pilha`

A implementação da classe `Pilha` esta ilustrada no código abaixo:

```
class Pilha: public LockManager{
protected:
 char op;
 int item;
 enum resultado_pilha resultado;
public:
 Pilha() : LockManager(ANDPERSISTENT)
 Boolean testeaceitacao() ;
 Boolean seletor(Pilha* resp[MAXVER]);
 char* marshalling() ;
 void unmarshalling(char* buff) ;
```

```
  virtual Boolean save_state(ObjectState& os, ObjectType ot;
  virtual Boolean restore_state(ObjectState& os, ObjectType ot) ;
  virtual  const TypeName type() const;
  virtual empilha(int item){};
  virtual desempilha(){};
  void set-outcome(enum stackOutCome outcome);
  void Message(const char* text);
};
```

O método `marshalling()` se encarrega de copiar o estado do objeto (o item a ser colocado na pilha e a operação ("um": `empilha()` ou "zero": `desempilha()`)) depois de executadas umas das operações desejada (`empilha()/desempilha()`) - para a mensagem (buffer) que será enviada ao servidor (Figura 4.10-a). O método `unmarshalling()` é responsável pela operação contrária, ou seja, copiar a mensagem recebida do servidor (contendo o resultado da exceção e o item) para o estado do objeto (Figura 4.10-b).

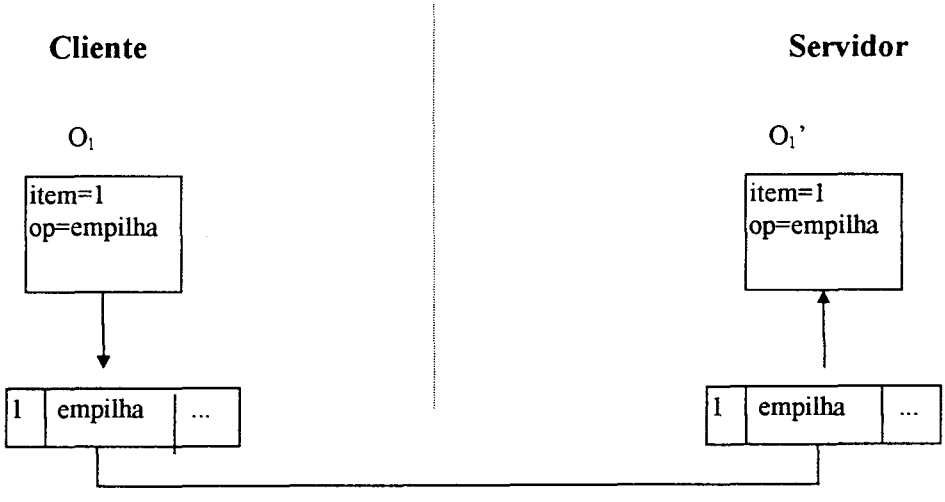



Figura 4.10-a: Transmissão de mensagem do cliente para o servidor

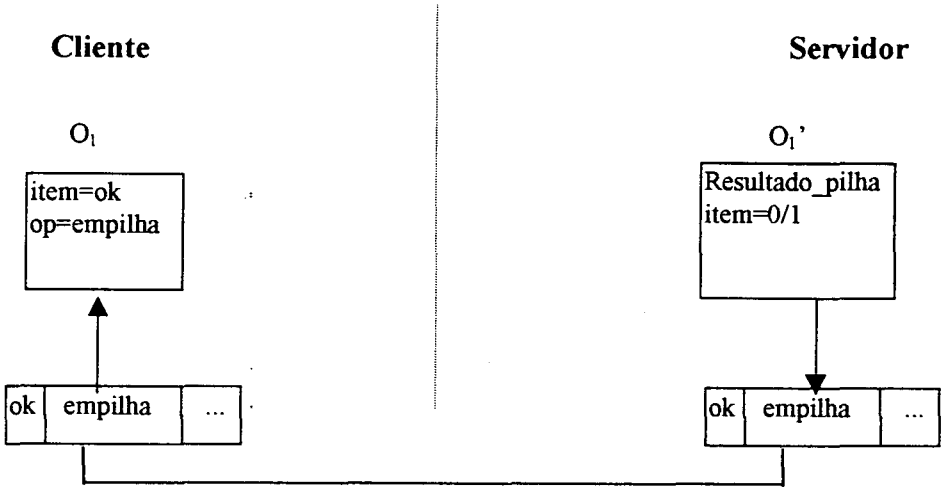



Figura 4.10-b: Transmissão de mensagem do servidor para o cliente

Os métodos `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()` (da classe `ObjectState`) para cada atributo que se deseja guardar. Analogamente, o método `restore_state()` deve conter uma chamada para o método `unpack()`(da classe `ObjectState`) para cada atributo salvo pelo método anterior. O método `type()` deve ser definido com a string que representa a hierarquia de herança do objeto.

O `testedeaceitacao()` e o `seletor()` são métodos implementados de forma a seguir uma propriedade definida pela especificação do projeto que é: "Todo elemento a ser inserido tem que ser igual ao seu índice". Para saber se essa propriedade foi validada ou não, quando as operações de `empilha()` ou `desempilha()` são executadas pelos servidores, eles retornam o resultado da pilha para exceção e um valor, que só pode ser 0/1 guardado no atributo item de cada um dos objetos. Se esse valor for igual a 0, significa que a propriedade foi violada, caso contrário, a propriedade se manteve e a variante passou no teste de aceitação e no seletor. A definição da implementação para os dois métodos (`testedeaceitacao()` e `seletor()`) são mostradas a seguir:

```
Bool Pilha::testedeaceitacao(){
 If (this->retorna_item() = = 1)
  Return TRUE;
Else  return FALSE;
}
```

```
Bool Pilha::seletor(Pilha* resp[MAXVER]){
Int certo=0;
Int errado = 0;
 For(int i=0; i<MAXVER;i++){
  If (resp[i]->retorna_item() = = 1)
   Certo++;
Else Errado++;}
If (certo > errado) return TRUE;
Else return FALSE;
}
```

### 4.2.2 Classe PilhaRobusta

Esta classe implementa e controla o uso de estratégias de tolerância a falhas para as variantes (`PilhaLista`, `PilhaVetor` e `PilhaDupla`). Ela é derivada da classe abstrata `Pilha` e se comporta como a classe paramétrica tolerante a falhas `TF`. Sua interface pública consiste das operações redefinidas `empilha()` e `desempilha()` e do construtor `PilhaRobusta()` que define os ponteiros para cada um dos objetos da aplicação.

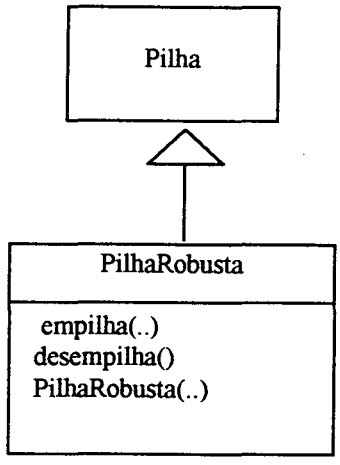



Figura 4.11 Classe `PilhaRobusta`

A implementação da classe `PilhaRobusta` está ilustrada no código abaixo:

```
class PilhaRobusta: public Pilha{
int Kind;
bool error;
TF<Pilha> pilha;
Int P_List;
Int P_Array;
Int P_Double;
public:
 PilhaRobusta(int k, int aux, int aux2, int aux3)
 empilha(int item);
 desempilha();
};
```

Os atributos privados P_List, P_Array e P_Double são valores inteiros que guardam o endereço da máquina servidora (port). A classe PilhaRobusta se comporta como uma classe tolerante a falhas.

### 4.2.3 Classe ExcPilha

A classe `ExcPilha` é uma classe que pode ser especializada pela aplicação pelas próprias classes do modelo para representar as exceções geradas durante a execução da aplicação Pilha. A classe `ExcPilha` representa as exceções de pilha que podem ser geradas durante a execução das variantes da aplicação, como por exemplo, exceções de pilha vazia, pilha cheia, etc, como mostra a Figura 4.12.

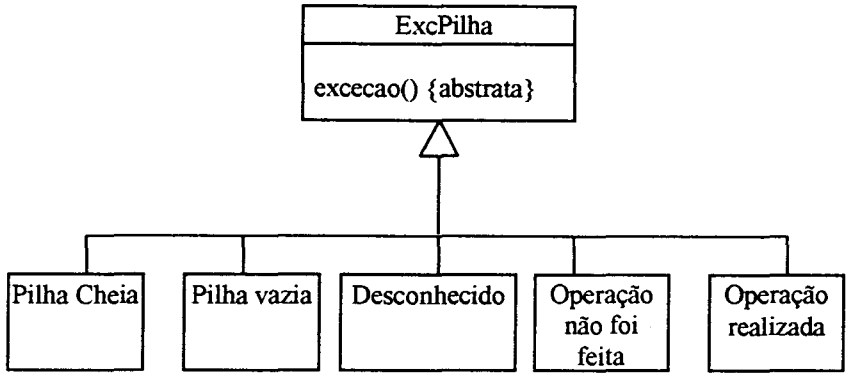



Figura 4.12: Classe `ExcPilha`

## 4.3 Validação e Avaliação do Framework

O framework proposto por essa dissertação faz testes com técnicas de injeção de falhas de software de forma a fazer uma análise de desempenho com os mecanismos de tolerância a falhas e tratamento de exceções. A técnica utilizada é o critério de Análise de Mutantes, onde erros são introduzidos nas variantes de forma a criar variantes com falhas, ou seja, mutantes. A seção 4.3.1 explicará o conceito da técnica de Análise de mutantes e como os mutantes foram criados para serem testados pelo framework; a seção seguinte fará uma avaliação de acordo com uma tabela de tempos de execução coletados para avaliar o desempenho das técnicas de tolerância a falhas.

### 4.3.1 Injeção de Falhas (Análise de Mutantes)

A injeção de falhas consiste em introduzir falhas num sistema de forma controlada. É muito usada na validação de mecanismos[29,49] de detecção e tratamento de erros em um sistema tolerante a falhas. É um nome geral dado a um número de técnicas usadas para acelerar a ocorrência de falhas, erros e defeitos em um sistema durante sua verificação dinâmica[49] ou seja, durante o seu tempo de execução. Um injetor de falhas é uma ferramenta que implementa essas técnicas.

A injeção de falhas pode adquirir diversas formas, segundo o nível de abstração utilizado para representar o sistema e o nível de aplicação das falhas. As formas mais comumente utilizadas são apresentadas em[29]: (i) simulação de falhas, (ii) injeção de falhas de software e (iii) injeção física de falhas. Na **simulação de falhas**, o sistema é representado por um modelo que caracteriza sua estrutura e/ou seu comportamento, onde falhas lógicas são introduzidas nesse modelo. Esta técnica abrange somente as fases de análise e projeto do sistema. Na **injeção física de falhas/injeção de falhas de hardware**, falhas físicas (permutação do bit em 0/1, curto circuito) são introduzidas em um protótipo de hardware e/ou software do sistema; podem ainda ser introduzidas diretamente sobre os pinos dos circuitos, ou submetendo o sistema a bombardeamento por íons pesados ou radiações eletromagnéticas, nesses casos, uma exceção é levantada. Neste tipo de técnica é utilizado hardware adicional para introduzir falhas dentro do hardware do sistema alvo.
Na **injeção de falhas de software**, erros são introduzidos com o objetivo de emular a conseqüência de falhas físicas no sistema. Esses erros consistem em alterações no conteúdo dos registros ou da memória, em alterações na seqüência de instruções (falhas de CPU), ou ainda na perda ou duplicação de mensagens (falhas no meio de comunicação). Nos últimos anos, pesquisadores têm preferido desenvolver ferramentas para injeção de falhas de software pois elas não requerem adição de hardware e também podem ser usadas em sistemas de aplicações e sistemas operacionais, o que é mais difícil fazer com injeção de falhas de hardware.

Um exemplo de testes realizados através da técnica de injeção de falhas de software em um sistemas tolerante a falhas pode ser encontrado em Zorzo et al. [64]. Neste trabalho,

uma avaliação de desempenho de uma aplicação reflexiva usando um mecanismo de injeção de falhas é apresentada. Nessa dissertação, a técnica de injeção de falhas de software utilizada é o Critério de Análise de Mutantes com o objetivo de avaliação de desempenho das técnicas de tolerância a falhas.

## Injeção de Falhas através do Critério de Análise de Mutantes

O critério Análise de Mutantes, segundo [59] é feito através da introdução de erros nos programas com pequenos desvios sintáticos, de forma a alterar a semântica do programa e, conseqüentemente, levar o programa a um comportamento incorreto. Existem ferramentas que auxiliam na aplicação do critério de Análise de Mutantes. Explicações mais detalhadas sobre essas ferramentas podem ser encontradas em[19,20,21].

Para ilustrar a aplicação do critério de Análise de Mutantes, considere que o programa da Figura 4.13.

```
Main(){  // Programa P
Int valor, num fat;
Fat=1;
Printf("Entre com o  numero e tecle <Enter>:");
Scanf("%d",&valor);
Num=valor;
If(num>=0) {
  While(num >1){
Fat = fat*num;
Num--;}
  Printf("O fatorial de %d e %d. \n", valor,fat);}
else
printf("Não existe fatorial de numero negativo!!\n");}
```

Figura 4.13: Programa Fatorial em teste

Os passos para Análise de Mutantes de acordo com o programa em questão são:

**Passo 1: Geração de Mutantes**:

Os mutantes são gerados através da aplicação de operadores de mutação do programa em teste (P). Operador de mutação são regras que definem as alterações que devem ser aplicadas no programa original P. A aplicação de um operador de mutação geralmente cria mais que um mutante, pois P pode conter várias entidades que estão no domínio da aplicação. As flechas das Figuras 4.14a e 4.1b indicam onde a inserção dos operadores de mutação ocorreu.

```
Main(){  // Programa P
Int valor, num fat;
Fat=1;
Printf("Entre com o  numero e tecle
    <Enter>:");
Scanf("%d",&valor);
Num=valor;
If(num>=0) {
  While(num >1){
Fat = (fat==num);
Num--;}
Printf("O fatorial de %d e %d. \n",
            valor,fat);}

else
 printf("Não existe fatorial de numero
     negativo!!\n");}
```

(a)

```
Main(){  // Programa P
Int valor, num fat;
Fat=1;
Printf("Entre com o  numero e tecle
    <Enter>:");
Scanf("%d",&valor);
Num=valor;
If(num<0) {
  While(num >1){
Fat = fat*num;
Num--;}
Printf("O fatorial de %d e %d. \n",
            valor,fat);}
else
 printf("Não existe fatorial de
numero negativo!!\n");}
```

b)

Figura 4.14: Mutantes gerados através dos operadores de mutação

**Passo 2: Execução do Programa**

O programa original P deve ser executado para os casos de testes selecionados, verificando sempre se o resultado é o esperado. Caso o programa apresente um resultado inesperado, então um erro foi descoberto e o programa termina. Entretanto, se a aplicação dos testes não revelar nenhum erro, então o passo seguinte é executado.

**Passo 3: Execução dos Mutantes**

Esse passo pode ser totalmente automatizado, ou seja, a intervenção de um testador não é necessária. O conjunto de testes T é aplicado a cada um dos mutantes criados. No nosso caso apenas dois mutantes serão executados. Se os resultados obtidos forem os mesmos significa que os mutantes são equivalentes (vivos) e isso pode ocorrer se:

- o conjunto de testes T não for suficiente para distinguir o comportamento de P;
- se, para qualquer dado do domínio de entrada, não existir diferença entre o comportamento dos dois programas.

Se, no entanto, os resultados comparados forem distintos, então o conjunto de testes T foi suficiente para expor as diferenças entre P e seus mutantes.

## Criação de Mutantes no Framework FOOD

No framework FOOD, as variantes são representadas pelas classes `PilhaVetor`, `PilhaLista` e `PilhaDupla`. O erro será introduzido alterando-se o valor de uma variável, chamada **status**, encapsulada em cada uma das classes variantes. O valor dessa variante é retornada para o cliente `PilhaRobusta` na chamada da operação `empilha()`.

Essa variável status indica a validade da propriedade de que todo elemento a ser inserido tem que ser igual a seu índice. A variável status possui dois valores: "zero": indica que houve erro na inserção do elemento, ou seja, o elemento foi inserido em uma posição diferente do seu índice; e "um": indica que a propriedade foi validada.

A implementação do método da classe `PilhaVetor` onde a variável será alterada é:

```
//Versão original

PilhaVetor::empilha(int item){
Array[free]=item;
If (item!=free)
  Status=0
Else Status = 1;
Free++
}
```

```
// Mutante

PilhaVetor::empilha(int item){
Array[free]=item;
If (item!=free)
  Status=0
Else Status = -1;
Free++
}
```

A implementação da operação da classe `PilhaLista` onde a variável será alterada é:

```
// Versão original
PilhaLista::empilha(int item, int i){
Node* aux = new node;
Aux->valor=item;
Aux->next = head;
Head=aux;
If(item != i)
  Status = 0;
Else Status = 1;
}
```

```
//Mutante
PilhaLista::empilha(int item, int i){
Node* aux = new node;
Aux->valor=item;
Aux->next = head;
Head=aux;
If(item == i)
  Status = 0;
Else Status = 1;
}
```

A implementação da operação da classe `PilhaDupla` onde a variável será alterada é:

```
//Versão Original
PilhaDupla::empilha(int item, int i){
Node* aux = new node;
Aux->valor=item;
Aux->next = head;
Aux->ant = NULL;
Head=aux;
If(item != i)
  Status = 0;
Else Status = 1;
```

```
//Mutante
PilhaDupla::empilha(int item, int i){
Node* aux = new node;
Aux->valor=item;
Aux->next = head;
Aux->ant = NULL;
Head=aux;
If(item != i)
  Status = 0;
Else Status = 0;
```

### 4.3.2 Avaliação do framework proposto

Esta seção mostra uma tabela com os tempos de execução para cada uma das técnicas (bloco de recuperação e n-versões) que foram coletados utilizando-se as variantes incorretas, ou seja, os mutantes. O objetivo é fazer uma análise de desempenho de acordo com os dados apresentados.

O tempo foi coletado em segundos, o tempo gasto para acesso em rede não foi desprezado, já que o exemplo foi implementado em um ambiente distribuído. O tempo gasto para o usuário foi considerado como uma constante para a execução de ambas as técnicas.

Os tempos gastos para o armazenamento de 100 elementos em uma pilha robusta segue os seguintes casos:

Para a técnica de N-versões:

1. Todas as variantes são executadas corretamente;
2. A 1ª variante é executada com falha e as outras variantes são executadas corretamente;
3. A 1ª e 2ª variantes são executadas com falhas e a 3ª variante é executada corretamente;
4. Todas as variantes são executadas com falhas.

| Framework Orientado a Objetos Distribuído - FOOD | | |
|---|---|---|
| Casos a serem avaliados para N-Versões | Tempo gasto(Seg) | Resultado do seletor |
| Todas as variantes são executadas corretamente | 22.03 |  |
| A 1ª variante é executada com falha e as outras variantes são executadas corretamente; | 21.60 |  |
| A 1ª e 2ª variantes são executadas com falhas e a 3ª variante é executada corretamente; | 22.03 |  |
| Todas as variantes são executadas com falhas. | 20.14 |  |

Figura 4.15: Tabela dos Tempos de Execução para N-Versões

Para a técnica de Bloco de Recuperação:

1. A 1ª variante é executada com sucesso;
2. A 1ª variante é executada com fracasso; e a 2ª variante é executada com sucesso;
3. A 2ª variante é executada com fracasso; e a 3ª variante é executada com sucesso;
4. A 3ª variante é executada com fracasso; portanto todas as variantes falharam ao executar.

| Framework Orientado a Objetos Distribuído - FOOD | | |
|---|---|---|
| Casos a serem avaliados para Bloco de Recuperação | Tempo gasto(Seg) | Resultado do teste de Aceitação |
| 1ª variante executada com sucesso. | 17.67 |  |
| 1ª variante falhou e 2ª variante sucesso | 22.06 |  |
| 1ª variante falhou, 2ª variante falhou e 3ª variante sucesso | 28.09 |  |
| Todas as variantes são executadas com falhas. | 31.02 |  |

Figura 4.16: Tabela dos Tempos de Execução para Bloco de Recuperação

Em ambas as tabelas apresentadas acima, notamos de imediato que o tempo de execução gasto pela técnica de N-versões é relativamente menor que o tempo de execução gasto pela técnica Bloco de Recuperação. Isso é lógico pois as variantes na técnica de N-versões são executadas em paralelo e elas não fazem uso das transações atômicas do Arjuna e nem do tempo gasto que o Arjuna utiliza para salvar o estado do objeto em disco, como é feito pelo Bloco de Recuperação. Os tempos coletados para a técnica de N-versões são quase que equivalentes.

Para a técnica de Bloco de recuperação os tempos coletados estão classificados em ordem crescente, devido ao fato de que cada vez que a restauração de estados é requisitada, o tempo gasto aumenta.

## 4.4 Sumário

O objetivo deste trabalho é fornecer uma hierarquia de classes que implementam os mecanismos de técnicas de tolerância a falhas de software acoplados com tratamento de exceções em sistemas distribuídos e que utilizam o contexto da programação orientada a objetos, de maneira a tornar efetiva a noção de reutilização de componentes. O framework proposto utiliza o ambiente de programação distribuída Arjuna e transações atômicas para implementar restauração de estados na técnica de bloco de recuperação fornecendo, assim, a recuperação de erros com retrocesso. O mecanismo de tratamento de exceção também é utilizado neste modelo como forma de prover recuperação de erros com avanço.

Os serviços do framework foram utilizados através de uma aplicação real tolerante a falhas e os tempos de execução coletados de acordo com a introdução de falhas nas variantes da aplicação de modo a fazer uma análise do desempenho das técnicas de tolerância a falhas: N-versões e Bloco de Recuperação. A injeção de falhas discutida e aplicada neste framework vem validar os mecanismos de tolerância a falhas de forma a mostrar o quanto eles são viáveis para o desenvolvimento de aplicações confiáveis.

# Capítulo 5

# Conclusões e Trabalhos futuros

## 5.1 Objetivo e desenvolvimento do trabalho

O objetivo deste trabalho é fornecer uma hierarquia de classes que implementam os mecanismos de técnicas de tolerância a falhas acoplados com tratamento de exceções em sistemas distribuídos utilizando técnicas orientadas a objetos, de maneira a tornar efetiva a noção de reutilização de componentes. O modelo proposto utiliza o ambiente de programação distribuído Arjuna e transações atômicas para implementar restauração de estados na técnica de bloco de recuperação fornecendo, assim, a recuperação de erros com retrocesso. O mecanismo de tratamento de exceção também é utilizado neste modelo como forma de prover recuperação de erros com avanço.

A injeção de falhas discutida e aplicada neste modelo vem validar os mecanismos de tolerância a falhas de forma a mostrar o quanto eles são viáveis para o desenvolvimento de aplicações confiáveis.

Uma outra característica importante da integração proposta vem do fato da utilização do paradigma orientado a objetos. Utilizando-se esse paradigma pode-se mostrar que exceções podem ser organizadas como uma hierarquia de classes, permitindo uma melhor estruturação dos códigos dos tratadores, simplificando ainda mais a tarefa de construir aplicações tolerantes a falhas.

O desenvolvimento deste trabalho envolveu as seguintes fases:

⇒ Adoção do modelo de objetos de forma a conseguir uma melhor estrutura para a construção de aplicações tolerantes a falhas;

⇒ Estudo do ambiente Arjuna de forma a utilizar suas funções para criar e controlar as ações atômicas;

⇒ Estudo de técnicas de Injeção de Falhas de forma a validar a aplicação gerando testes que comprovassem os mecanismos de tolerância a falhas utilizados;

## 5.2 Trabalhos Relacionados

Esta seção discute alguns trabalhos relacionados com o desenvolvimento de software tolerante a falhas encontrados na literatura.

Poucos trabalhos são encontrados na literatura que exploram as técnicas orientadas a objetos para o desenvolvimento de software tolerante a falhas. Um deles é proposto por Rubira e Stroud[51] onde a recuperação de erros com avanço e com retrocesso são implementadas utilizando-se a linguagem de programação C++ envolvendo programação seqüencial. Outro trabalho, também de Rubira[52] apresenta dois esquemas genéricos sob o paradigma de objetos que dão suporte à implementação de componentes ideais com diversidade de projeto.

O trabalho de Xu et al.[61] incorpora os mecanismos complementares de conversação[48] ( usada para prover recuperação de erros coordenada entre as atividades envolvendo processos concorrentes) e transação[31] (usada para manter a consistência do compartilhamento de recursos na presença de acessos concorrentes) para implementar software tolerante a falhas.

O trabalho de Randel e Xu[45] descreve um framework abstrato para expressar esquemas de tolerância a falhas de software em ambientes centralizados. Esse framework é composto por três classes distintas: **Variantes, Seletores e Arquitetura de Controle**. A classe abstrata Variante é responsável por definir uma interface através da qual o programador da aplicação pode desenvolver variantes de programas concretos. Ela possui um conjunto de tratadores de exceção padrão (endereço fora dos limites, divisão por zero) que tratam as exceções locais que podem surgir durante a execução das variantes definidas pelo usuário. A classe Arquitetura de Controle organiza a execução das variantes definidas

pelo usuário de forma a prover ou redundância estática ou redundância dinâmica. A classe abstrata Seletor define a implementação da operação de teste de aceitação e seletor e provê tratadores de exceções para exceções que podem surgir durante a execução desses dois métodos.

O framework proposto nessa dissertação foi construído para se adaptar em ambientes distribuídos , ao contrário do modelo proposto por Randel e Xu [45], por isso as variantes do nosso framework são modeladas como servidores e as informações referentes à aplicação, como por exemplo, o teste de aceitação, são encapsuladas no próprio objeto.

Um trabalho complementar é o de Corrêa[14], onde o mesmo framework é empregado utilizando-se o conceito de reflexão computacional. Nesse trabalho um framework FOORD (Framework Orientado a Objetos Reflexivo e Distribuído) é implementado de forma que as aplicações tolerantes a falhas sejam organizadas em níveis visando a reutilização e a transparência de serviços, contribuindo para mostrar como a técnica de reflexão computacional pode ser viável e útil no desenvolvimento de aplicações tolerantes a falhas. Alguns trabalhos já foram propostos para explorar reflexão computacional e orientação a objetos para tolerância a falhas de software[14,42,50,62]. Xu et al.[60] apresenta uma idéia de uma arquitetura reflexiva para tolerância a falhas, mas resultados concretos não são mostrados.

## 5.3 Contribuições

Entre as contribuições feitas através do desenvolvimento dessa dissertação, podemos listar:

- ⇒ Implementação de um framework em C++ que dá apoio ao desenvolvimento de aplicações tolerantes a falhas de software em ambientes distribuídos;
- ⇒ Incorporação do mecanismo de tratamento de exceções em um ambiente distribuído.

## 5.4 Extensões Futuras

No framework descrito nessa dissertação, duas técnicas de tolerância a falhas são providas: Bloco de recuperação e N-versões. Essas duas técnicas garantem a tolerância a falhas de software e são compostas por variantes e pelas operações de teste de aceitação e seletor, respectivamente. Mesmo garantindo a tolerância de falhas de software de uma aplicação, falhas podem ocorrer na implementação dos métodos e classes considerados críticos do framework, que são: teste de aceitação, seletor, bloco de recuperação e n-versões. Bloco de recuperação e N-Versões são classes genéricas do framework que podem ser reutilizadas, por isso é importante que essas classes estejam livres de erros.

Portanto, uma extensão para o nosso trabalho seria a realização de testes exaustivos sobre essas implementações de maneira a reduzir a possibilidade de ocorrerem falhas durante as suas execuções. Uma outra área para extensão de nosso trabalho seria utilizar uma linguagem de programação com suporte para interfaces gráficas, como Java, que pudesse dar apoio às interações usuário-máquina facilitando o uso do framework.

# Apêndice A

A .1 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 TF, Pilha

END_CLASS LockManager

A .2 Classe Pilha

CLASS Pilha

DESCRIPTION Pilha é uma classe abstrata que define a raiz da hierarquia para objetos tolerantes a falhas da aplicação. As variáveis protegidas op, item e o resultado guardam respectivamente: a operação da pilha(empilha ou desempilha, o elemento a ser empilha e/ou desempilhado, e o resultado que informa o tipo de exceção sinalizada.

---

Op

Item

Resultado_pilha

---

PROVIDED_INTERFACE

OPERATIONS [Métodos públicos definidos pela classe]

**Boolean seletor(Pilha* resp[])**

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

**Boolean testeaceitacao()**

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

**char* marshalling()**

DESCRIPTION Este método deve ser definido pela aplicação para implementar o empacotamento dos estados do objeto para serem enviados 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 void empilha()**

DESCRIPTION Este método deve ser definido pela aplicação para empilhar um elemento.

**virtual int desempilha()**

DESCRIPTION Este método deve ser definido pela aplicação para desempilhar um elemento

SUPERCLASSES LockManager

SUBCLASSES classe da aplicação (Pilha Robusta) que implementa os serviços críticos e classes que representam suas variantes (PilhaVetor, PilhaLista e PilhaDupla).

END_CLASS Pilha

A .3 Classe ExcSocket

CLASS ExcSocket

DESCRIPTION A classe ExcSocket é uma classe abstrata que pode ser especializada pela aplicação pelas próprias classes do framework para representar as exceções de Socket 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 Criação, Conexão, Host, Escrita, Leitura, Ligação, NomeSocket e Aceitação.

END_CLASS ExcSocket

A .4 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 Socket

A .5 Classe ExcPilha

CLASS ExcPilha

DESCRIPTION Esta classe representa as exceções da pilha que podem ser geradas durante a execução das variantes da aplicação

PROVIDED_INTERFACE

OPERATIONS [Métodos públicos definidos pela classe]

**virtual int exceção()**

DESCRIPTION Este retorna o tipo de exceção da pilha gerada.

SUPERCLASSES Não possui SuperClasse

SUBCLASSES Pilha Cheia, Pilha Vazia, Desconhecido, Operação não foi Feita, Operação realizada.

END_CLASS ExcPilha

A .6 Classe PilhaRobusta

CLASS PilhaRobusta

DESCRIPTION PilhaRobusta é uma classe concreta derivada da classe abstrata Pilha e que implementa as técnicas tolerantes a falhas da aplicação por se comportar como uma classe tolerante a falhas. As variáveis Kind, P_List, P_Array e P_Double guardam respectivamente: o tipo de técnica: bloco de recuperação ou n-versões, endereço para o servidor da lista, endereço para o servidor do array e o endereço para o servidor da PilhaDupla.

---

Kind

P_List

P_Array

P_Double

---

PROVIDED_INTERFACE

OPERATIONS [Métodos públicos definidos pela classe]

PilhaRobusta(int k, int aux1, int aux2, int aux3)

DESCRIPTION Este método é o construtor do objeto de PilhaRobusta, inicializando os endereços dos servidores para as variantes.

**void empilha(int item)**

DESCRIPTION Este método deve ser definido pela aplicação para empilhar um elemento.

**int desempilha()**

DESCRIPTION Este método deve ser definido pela aplicação para desempilhar um elemento

SUPERCLASSES Pilha

END_CLASS PilhaRobusta

## Referências


[1] T. Anderson — *Fault Tolerant Computing. Resilient Computing Systems*, T. Anderson(Ed.), capítulo 1, Collins Professional and Technical Books, 1985.

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

[3] H. E. Bal and D. Grune — *Programming Language Essentials.* Addison-Wesley.

[4] R. Balter, S. Lacourte, and M. Riveill. — The Guide language, *The Computer Journal*, 7(6):519-530, 1994.

[5] D. M. Beder — Integração dos Mecanismos de Recuperação de Erros por Avanço e por Retrocesso. Dissertação de Mestrado. Instituto de Ciência da Computação. Universidade Estadual de Campinas. Maio 1997.

[6] D. M. Beder, L.E. Buzato e C.M.F.Rubira. — Exceções e a construção de programas confiáveis. *Anais do II Simpósio Brasileiro de Linguagens de Programação*, Campinas-SP, 3 a 5/11/97,editores. L.E. Buzato e C. M. F. Rubira, pp 231-244.

[7] M. Benveniste and V. Issarny. — Concurrent programming Notations in Object-oriented Language Arche. *Technical report*, IRISA/INRIA-Rennes, França Dezembro 1992.

[8] G. Blair, J. Gallagher, D. Hutchison and D. Sheperd. — *Object-Oriented Languages, Systems and Applications.* G. Blair et al(ed.), Capítulo 4, Pitmam, 1991.

[9] A. Borgida — Exceptions in Object-oriented Languages. *SIGPLAN NOTICES*, 21(10), Outubro 1986.

[10] A. Burns and A. Wellings — *Real-Time Systems and their programming languages.* Addison-Wesley.

[11] C.M.F.R. Calsavara and R.J. Stroud. — Forward and Backward Error Recovery in C++. *Technical Report Series nº 417.* University of Newcastle Upon tyne, UK, Fevereiro 1993.

[12] L. Cardelli and P. Wegner. — On Understanding Types, Data Abstraction, and Polimorfism. *ACM Computing Surveys*, 17(4)):480-521, Dezembro 1985.

[13] R. Chillarege. Challenges Facing Software Fault-Tolerance. *Research Report*, R.C. 20281(89648), IBM Research Division, T.J. Watson Research Center, Novembro 1995.

[14] Sand Luz Corrêa Implementação de Sistemas Tolerantes a Falhas usando Programação Reflexiva orientada a Objetos. Dissertação de Mestrado. Instituto de Ciência da Computação. Universidade Estadual de Campinas. Dezembro 1997.

[15] F. Cristian. Exception Handling and Software fault Tolerance. *IEEE Transactions on Computers*, C-31(6):531-540, Junho 1982.

[16] F. Cristian. Exception Handling. *Computer Science and Engineering Department.* University of California, San Diego, La Jolla, Technical Report n° CA 92093-0114.

[17] Q. Cui and J. Gannon. Data-oriented Exception Handling. *IEEE Transactions on Software Engineering.*, 18(5):393-401, Maio 1992.

[18] M. Day, R. Gruber, B. Liskov and A.C. Myers. Subtypes vs. Where clauses: Constraining Parametric Polymorphism. *Proc. OOPLSA'95*, 30(10), Outubro 1995.

[19] M. E. Delamaro "Proteum – Um ambiente de testes baseado em Análise de Mutantes". Dissertação de Mestrado, ICMSC/USP – São Carlos, SP. Outubro 1993.

[20] M. E. Delamaro; J.C. Maldonado e A. P. Mathur Integration Testing Using Interface Mutation. *In Proceedings of the VII International Simposyum of Software Relibility Engineering(ISSRE)*, New York, Novembro, 1996.

[21] R.A. DeMilo et al. Na Extended Overview of the Mothra Testing Environment. *In Proc. Of the Second Workshop on Software testing, Verification and Analysis.* Banff-Canadá, 1988.

[22] Department of Computing Sciene, University of Newcastle upon Tyne. The Arjuna System programmers's Guide, public release 3.0 edition, 1994.

[23] S.J. Drew and K.J. Gough. Exception Handling: Expecting the Unespected. *Comput. Lang.*, 32(8):69-87, 1994.

[24] C. Ghezzi and M. Jazayeri. *Programming languages Concepts*, 3ª Edição, John Wiley, 1997.

[25] J.B. Goodnough. Exception handling: Issues and a proposed Notation. *Communications of the ACM*, 18(12):683-696, Dezembro 1975

[26] Neil Graham. *Learning C++*. McGraw-Hill International Editions. Computer Science Series, 1991.

[27] S.T. Gregory and J.C. Knight. A New linguistic Approach to Backward Error Recovery. *Em 15th International Symposium on Fault tolerant Computing Systems*, pp: 404-409, 1985.

[28] J.J. Horning , H.C. Lauer, P.M. Melliar-Smith and B. Randell. A Program structure for Error Detection and Recovery. *Lecture Notes in Computer Sciene*, 16:171-187, 1974.

[29] M.C. Hsueh, T. K. Tsai e R. K. Iyer Fault Injection Techniques and tools. . *IEEE Transactions on Software Engineering*, SE-1(2):75-82, Abril, 1997.

[30] P. Jalote and R.H. Campbell. Atomic Actions for Fault-Tolerance Using CSP. *IEEE transactions on Software Engineering*, SE-12(1):59-68, Janeiro 1986.

[31] P. Jalote. *Fault Tolerance in Distributed System*. Prentice-hall, Inc., 1994

[32] A. Koening An Enception Ideal. *Journal of Object-Oriented Programming*, 3(2):52-59, 1990.

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

[34] R. Koo and T. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1):23-31, Janeiro 1987.

[35] S. Krakowiak, M. meysembourg, H. nguyen, M. Rivelli, C. Roisin and X. Rousset de Pina. Design and Implementation of na Object-Oriented, Strongly Typed language for Distributed Applications. *Journal of Object-Oriented Programming*, 3(3), Setembro 1990.

[36] S. Lacourte. Exceptions in Guide, an Object-Oriented language for Distributed Applications. *Lectures Notes In Computer Sciene*, 512:268-287, Julho 1991.

[37] P.A. Lee Software-Faults: The Remaining Problem in Fault Tolerant Systems? *Hardware&Software Arch. For* FT,

LNCS # 774, Springer-Verlag, 1994.

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

[39] B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300-312, Março 198.

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

[41] B.H. Liskov and A. Snyder. Data Abstractions and Hierarchy. *ACM SIGPLAN Notices*, 23(5):17-34, Novembro 1988.

[42] M.L.B. Lisboa, C.M.F. Rubira and L.E. Buzato Arquitetura reflexiva para o Desenvolvimento de Software Tolerante a Falhas. *SEMISH'96*, Recife, PE. Agosto 1996, pp: 155-165.

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

[44] J. M. Purtillo and P. Jalote. An Environment for Developing Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, 17(2), pp: 153-159, Fevereiro 1991.

[45] B. Randell and J. Xu. Object-Oriented Software Fault Tolerance: Framework, Reuse and Design Diversity. *Em 1st PDCS2 Open WorkShop*, pags 165-184, Toulouse, 1993.

[46] B. Randell and J. Xu. Recovery Blocks. *Technical Report Series nº 479*. University of Newcastle upon Tyne, UK, Abril 1994.

[47] B. Randell, A. Romanovsky, C. Rubira, R. Stroud, Z.Wu. and J. XU. *From Recovery Blocks to Coordinated Atomic Actions, em: Predictably Dependable Comp. Syst.*, Eds. Randell et al., Springer-Verlag, pp:87-101, 1995.

[48] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220-232, 1975.

[49] H.A. Rosemberg and K.G. Shin Software Fault Injection and its application in Distributed Systems. *Proc. FTSC-23*, Toulouse, França, 1993.

[50] C.M.F. Rubira, S.L.Correa and L.E. Buzato Um Framework Orientado a Objetos Reflexivo para a Construção de Software Tolerante a Falhas. . *Em I Simpósio regional de Tolerância a Falhas*, pags 75-89,

Porto Alegre, Rio Grande do Sul, Brasil, Dezembro 1996.

[51] C.M.F.Rubira and R.J. Stroud. Forward and Backward Error Recovery in C++.*Journal of OO Systems*, 1(1),pp:61-85, Outubro 1994.

[52] Cecilia M.F. Rubira *Structuring Fault-Tolerant Object-Oriented Systems Using Inheritance and Delegation.* PhD Thesis. Department Of Computing Science. University of Newcastle Upon Tyne, Outubro 1994.

[53] J. Rumbaugh, M. Blaha, and W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.Prentice-Hall*, Englewood Cliffs, NJ, USA, 1991.

[54] Robert W. Sebesta. Exception Handling. In *Concepts of Programming languages,* Cap. 12, pags. 380-399, Robert W. Sebesta, Benjamim/Cummings, 1991.

[55] S.K. Shrivastava, G.N. Dixon, and G.D. parrington. An Overview of the Arjuna Programming System. *IEEE Software,* 8(1):66-73, janeiro 1991.

[56] S.K. Shrivastava, L.V. Mancini, and B. Randell. The Duality of Fault-Tolerant System Structures. Software-*Practice and Experience*, 23(7):173-182, Fevereiro 1993.

[57] S.K. Shrivastava. Concurrent Pascal with backward Error Recovery. *Software Practice and Experience*, 9:1021-1033, 1979.

[58] B. Stroustrup. *The C++ Programming language*. Addison Wesley, 2ª edição, 1991.

[59] A. M. R. Vicenzi et al. Critério Análise de Mutantes: Estado Atual e Perspectivas. *ANAIS do Workshop do Projeto validação e teste de Sistemas de operação.* Águas de Lindóia- S.P., janeiro-1997, pag:15-26.

[60] J. Xu, B. Randell, A. Romanovsky, C.M.F. Rubira and R. Stroud. Toward an OO Approach to Software Fault Tolerance. *In Fault-Tolerant Par. And Distr. Systems,* D.R. Averesky(Ed.), IEEE Comp. Soc. Press, Dezembro 1994.

[61] J. Xu, B. Randell, A. Romanovsky, C.M.F. Rubira, R. Stroud, and Z.Wu. Fault Tolerance in Concurrent Object-Oriented Software through Coodinated Error Recovery. *Em FTCS-25: 25th International Symposium on Fault Tolerant Computing Digest of Papers*, pags 499-509, Pasadena, Califórnia, 1995.

[62] J. Xu, B. Randell, C.M.F. Rubira-Calsavara, R.J. Software Fault Tolerance: Forwards na Object-oriented Approach. *Technical Report Series nº 498.* University of

Stroud. Newcastle Upon Tyne, UK, Novembro 1994.

[63] S. Yemini and D.M. Berry. A Modular Verifiable Exceptions Handling Mechanism. *ACM Transactions on programming languages and Systems*, 7(2):214-243, Abril 1985.

[64] A. Zorzo, J.Xu and B.Randell. Experimental Evaluation of fault-Tolerant Mechanisms OO Software. *SEMISH'96*, Recife, PE. Agosto 1996, pp: 457-468.

[65] R.E. Johnson and B. Foote Designing Reusable Classes. *Journal of object oriented Programming* – JOOP, 1(2):22-35, junho/Julho 1988.