Uma Abordagem Orientada a Objetos
para Programação Distribuída Confiável

Elbson Moreira Quadros

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Uma Abordagem Orientada a Objetos para Programação Distribuída Confiável

Elbson Moreira Quadros

Junho de 1997

Banca Examinadora:

- Profa. Dra. Cecília Mary Fischer Rubira (Orientadora)
  Instituto de Computação — UNICAMP
- Profa. Dra. Taisy Silva Weber
  Instituto de Informática — UFRGS
- Prof. Dr. Luiz Eduardo Buzato
  Instituto de Computação — UNICAMP
- Prof. Dr. Hans Kurt E. Liesemberg
  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

Uma Abordagem Orientada a Objetos para Programação Distribuída Confiável

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida por Elbson Moreira Quadros e aprovada pela comissão julgadora.

Campinas, 09 de junho de 1997.

Cecília Mary Fischer Rubira

Cecília Mary Fischer Rubira
(orientadora)

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

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Quadros, Elbson Moreira

Q22a Uma abordagem orientada a objetos para programação distribuída confiável / Elbson Moreira Quadros -- Campinas, [S.P. :s.n.], 1997.

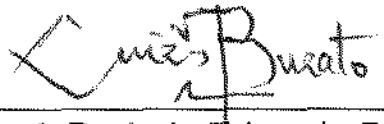
Orientador : Cecília Mary Fischer Rubira

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

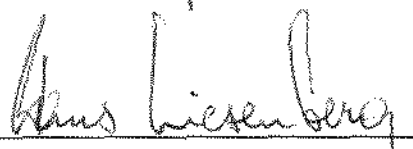
1. Programação orientada a objetos (Computação). 2. Computação tolerante de falhas. I. Rubira, Cecília Mary Fischer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Tese de Mestrado defendida e aprovada em 09 de junho de 1997 pela Banca Examinadora composta pelos Professores Doutores

Profª. Drª. Taisy Silva Weber



Prof. Dr. Luiz Eduardo Buzato



Prof. Dr. Hans Kurt Edmund Liesenberg

Profª. Drª. Cecília Mary Fischer Rubira

À memória de minha avó
Ana Maria de Jesus

Resumo

Este trabalho tem por objetivo aplicar técnicas de orientação a objetos para estruturar aplicações complexas, visando obter uma melhoria da qualidade e confiabilidade dessas aplicações. Várias técnicas orientadas a objetos são exploradas, tais como: abstração de dados, compartilhamento de comportamento (incluindo herança e delegação), classes abstratas, polimorfismo e acoplamento dinâmico. Nós propomos a utilização dessas técnicas na estruturação de aplicações distribuídas, provendo suporte para tolerância a falhas de ambiente através da incorporação disciplinada de redundância, de forma que o impacto dessa redundância na complexidade do sistema possa ser mantido sob controle. Para o entendimento e validação dessas técnicas foi desenvolvido um protótipo de uma aplicação distribuída orientada a objetos: um Controlador de Trens. Além disso, utilizamos duas abordagens promissoras para reutilização de software em grande escala — padrões de projeto e metapadrões — para a construção de um *framework* orientado a objetos para o subdomínio de controladores de trens.

Abstract

The major goal of this work is to apply object-oriented techniques for structuring complex object-oriented applications, and to relate them to the improvement of quality and reliability of large computer applications. We use a collection of object-oriented concepts, features and mechanisms, such as data abstraction, inheritance, delegation, abstract classes, polymorphism and dynamic binding. We propose an approach for the provision of environmental fault tolerance and distribution, based on the incorporation of redundancy in an incremental way, so that the complexity can be kept under control. In addition, we show how such techniques can be used to develop reusable and easier to extend software components. For the understanding and validation of these techniques, we developed a prototype of an object-oriented distributed and dependable railway controller application.

Besides, we use design patterns and metapatterns — two promising approaches for software reuse — for developing an object-oriented framework for a railway controller subdomain.

Agradecimentos

À Cecília M. F. Rubira, minha orientadora, pela maneira coerente e prestativa com que conduziu este trabalho.

À minha família, sobretudo, meus pais (Erivaldo e Senhorinha) e meus irmãos (Sueli e Elton), pelo grande apoio que sempre me deram e pela confiança em mim depositada.

À Nádia, minha namorada, pelo carinho, compreensão e cooperação a mim concedidos.

A todos os meus amigos, em especial, Célio Targa, Daniel Paiva, Delano Beder, Mauro Oliveira, Rodrigo Netto e Solange Sobral, pelo companheirismo e convivência mais que agradável.

Aos professores e funcionários do Instituto de Computação — UNICAMP, pelos serviços prestados ao programa de mestrado.

À Fundação de Amparo à Pesquisa do Estado de São Paulo — FAPESP (Processo 96/2705-4) e ao Conselho Nacional de Desenvolvimento Científico e Tecnológico — CNPq (Processo 130278/95-5), pelo apoio financeiro.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Muitos sistemas de computação são bastante complexos como, por exemplo, sistemas reativos que controlam processos físicos; aplicações que mantêm a integridade de milhares de registros de informação ao mesmo tempo que atualizações e consultas concorrentes são realizadas; e sistemas para o controle e monitoração de entidades do mundo real, tais como, controle de tráfego aéreo ou ferroviário. Esses sistemas têm-se tornado mais complexos ainda devido aos novos requisitos das aplicações, tais como, confiabilidade, segurança, alta disponibilidade, etc. Além disso, a demanda por redes de computadores e recursos distribuídos introduziu novos níveis de complexidade na construção desses sistemas.

O principal objetivo deste trabalho é aplicar técnicas de orientação a objetos para estruturar aplicações complexas, visando obter uma melhoria da qualidade e confiabilidade dessas aplicações. Nós propomos a utilização de técnicas de orientação a objetos na estruturação de aplicações distribuídas, provendo suporte para tolerância a falhas através da incorporação disciplinada de redundância, de forma que o impacto dessa redundância na complexidade do sistema possa ser mantido sob controle. Nós mostramos também como estas técnicas podem ser utilizadas para construir componentes de software reutilizáveis e de fácil extensão.

O paradigma de orientação a objetos tem sido considerado um dos mais significantes progressos para o desenvolvimento de sistemas de computação [Rum92, Boo91, Rub94]. Este paradigma é adequado para estruturar uma grande variedade de sistemas complexos, provendo técnicas para desenvolver aplicações que são mais fáceis de manter e reutilizar.

Uma característica do modelo de objetos é a individualidade: objetos geralmente são concebidos como unidades singulares de serviços. A computação é organizada através de objetos agrupados em classes similares, com comportamentos próprios, que interagem com outros objetos para solicitar ou fornecer serviços. Esta visão de programas compostos por objetos que cooperam entre si e de objetos como prestadores de serviços é bastante conveniente para se expressar um modelo distribuído e tolerante a falhas. Para tanto, vários conceitos e técnicas orientados a objetos foram explorados, tais como: abstração de dados, compartilhamento de comportamento (incluindo herança e delegação), classes abstratas, polimorfismo e acoplamento dinâmico. Além disso, utilizamos três abordagens promissoras para reutilização de software em grande escala — padrões de projeto, metapadrões e *frameworks* — que são baseadas no modelo de objetos.

Para o entendimento e validação dessas técnicas foi desenvolvido um protótipo de uma aplicação distribuída orientada a objetos: um Controlador de Trens.

1.1 Descrição do Problema

O problema que nós propomos resolver caracteriza-se por dois requisitos principais: tolerância a falhas de ambiente e recursos distribuídos através de uma rede de computadores.

Em muitas situações do mundo real, entidades no domínio do problema podem mudar seus comportamentos devido a uma grande variedade de fenômenos como, por exemplo, defeitos de equipamentos, erros de pessoas, falhas de sensores, etc. Estas mudanças podem ser classificadas de várias maneiras, tais como: freqüentes e infreqüentes, desejáveis e indesejáveis, previsíveis e imprevisíveis. As mudanças infreqüentes, indesejáveis e imprevisíveis no comportamento das entidades do ambiente são denominadas falhas de ambiente. Um sistema robusto e confiável deve ser capaz de tolerar tais falhas, além das falhas de hardware e software. Um dos requisitos do problema refere-se a estas entidades do mundo real com as quais o software tem que interagir e que possuem não apenas uma fase normal de comportamento mas também uma ou mais fases anormais de comportamento associadas à ocorrência de diferentes falhas de ambiente.

Um outro requisito refere-se à distribuição dos recursos do sistema através de uma rede de computadores. Nos últimos anos, as organizações passaram a explorar sistemas de computação mais efetivamente e isto proporcionou o surgimento de aplicações distribuídas de grande porte. Estas aplicações oferecem uma computação altamente integrada para usuários que podem estar fisicamente separados e interagindo com uma multiplicidade de componentes de uma rede de computadores. O desenvolvimento de software para tais plataformas implica em grandes desafios, pois o sistema precisa comportar-se de maneira consistente, independente do local de acesso. O nosso trabalho está mais especificamente relacionado com dois aspectos: particionamento e alocação de componentes de software nos nodos físicos da rede e separação entre a funcionalidade das aplicações dos mecanismos que implementam distribuição.

1.2 Solução Proposta

Em nosso modelo, tolerância a falhas de ambiente é provida com base nas estruturas propostas em [Rub94]. A idéia chave consiste em identificar os objetos da aplicação que têm seu comportamento alterado em tempo de execução devido a mudanças no ambiente externo. Em seguida, é construída uma hierarquia de classes que captura as diferentes fases do comportamento destes objetos. Esta hierarquia é construída em paralelo com a hierarquia de classes da aplicação propriamente dita. As duas hierarquias são ligadas por um mecanismo de delegação.

O problema de particionamento dos componentes de um sistema distribuído através de uma rede de computadores é "naturalmente" suportado pelo modelo de orientação a objetos. Para tanto, utilizamos conceitos como o de objetos e o de composição de objetos para representar as unidades de distribuição. Além disso, para resolver o problema da separação da funcionalidade da aplicação dos mecanismos que implementam distribuição, definimos um objeto intermediário (conhecido como *proxy*), que implementa o protocolo de comunicação entre os objetos distribuídos e, com isso, permite o acesso transparente a um objeto em outro espaço de endereçamento.

Por fim, com o objetivo de aproveitar todo o potencial da construção de software orientado a objetos, estudamos o conceito de arquiteturas reutilizáveis. Nesse aspecto, a

nossa abordagem consiste em fatorar as estruturas de classes da aplicação, formando um conjunto de classes abstratas e classes concretas interligadas entre si, visando a criação de um *framework* orientado a objetos, isto é, um conjunto de classes, que provê uma infra-estrutura genérica de soluções para um conjunto de problemas. Para a estruturação deste *framework* foram empregados os conceitos de padrões de projeto e metapadrões.

1.3 Contribuições

A principal contribuição do nosso trabalho decorre da experiência prática em aplicar técnicas de orientação a objetos para estruturar programas distribuídos e confiáveis. Este trabalho tem por objetivo propor discussões e sugestões, com base nos resultados práticos da aplicação e combinação de uma série de técnicas existentes; algumas já bastante difundidas (como por exemplo, herança e delegação) e outras recentes (como por exemplo, padrões de projeto e metapadrões), mas muito promissoras. Podemos, portanto, destacar as seguintes contribuições:

- desenvolvimento de um protótipo para uma aplicação distribuída e confiável utilizando uma abordagem orientada a objetos, em particular, explorando o mecanismo de delegação;
- estruturação de um *framework* orientado a objetos com base no conhecimento do domínio do protótipo desenvolvido;
- análise comparativa entre o uso de padrões de projeto e metapadrões; duas abordagens utilizadas para construção de sistemas reutilizáveis;
- utilização prática da metodologia UML, que representa a unificação das metodologias de Booch, Rumbaugh (OMT) e Jacobson (OOSE). Esta metodologia foi utilizada para a modelagem do protótipo e também para a modelagem do *framework*;

1.4 Organização da Dissertação

O restante desta dissertação é organizado da seguinte maneira:

Capítulo 2 descreve os principais conceitos do paradigma de orientação a objetos e apresenta a terminologia utilizada nesta dissertação. Descreve uma abordagem para modelar estados através de hierarquia de classes. Introduz a metodologia de análise e projeto adotada nesta dissertação. Introduz também os conceitos de *frameworks*, padrões de projeto e metapadrões, que são técnicas promissoras para a reutilização de software em grande escala.

Capítulo 3 apresenta uma série de definições e conceitos básicos de tolerância a falhas e sistemas distribuídos. Descreve tolerância a falhas, incluindo alguns comentários sobre mecanismos para tolerância a falhas de hardware, falhas de software e falhas de ambiente. Descreve sistemas distribuídos, incluindo a descrição do modelo computacional, de mecanismos para comunicação entre processos e, também, alguns comentários sobre a integração de sistemas distribuídos com o paradigma de orientação a objetos.

Capítulo 4 apresenta um estudo de caso detalhado para aplicar, de maneira prática, as várias técnicas estudadas. Este estudo de caso inclui a especificação, os requisitos e o modelo de análise/projeto de uma aplicação para controle e monitoração de trens em um modelo de ferrovia.

Capítulo 5 estende a aplicação descrita no estudo de caso do capítulo 4, com o objetivo de discutir reutilização de software em grande escala. Esta extensão é provida através da estruturação de um *framework* para o domínio de Controladores de Trens. O *framework* é proposto com base na integração dos conceitos de padrões de projeto e metapadrões.

Capítulo 6 apresenta alguns trabalhos relacionados. Propõe extensões e trabalhos futuros, e apresenta algumas conclusões.

Capítulo 2

Fundamentos de Orientação a Objetos

O paradigma de orientação a objetos descreve um estilo de programação baseado nas noções de classes, objetos e herança. Uma classe descreve um conjunto de objetos com a mesma estrutura e o mesmo comportamento. O comportamento de um objeto é caracterizado pela sua interface, ou seja, o conjunto de operações e atributos públicos da sua classe. Um objeto é composto de atributos e de operações que são realizadas sobre esses atributos. Ao receber uma mensagem, o objeto executa a operação solicitada, através de computações sobre seus próprios atributos e, possivelmente, requisitando a execução de operações de outros objetos. Herança é um mecanismo para derivar novas classes a partir de classes já existentes através de um processo de refinamento. Uma classe derivada herda a representação dos atributos e operações da classe base, mas pode adicionar novas operações, estender a representação de atributos ou sobrepor a implementação de operações já herdadas. Deste modo, a programação orientada a objetos consiste na definição de classes e na criação de hierarquias de classes, onde as propriedades comuns podem ser transmitidas das superclasses para as subclasses através do mecanismo de herança.

O crescente interesse por esse paradigma se deve ao fato dele prover uma maneira melhor de estruturar e construir sistemas complexos, uma vez que ele facilita a reutilização, modificação e extensão dos componentes de software, melhorando a qualidade e reduzindo o custo de desenvolvimento [Rum92, Boo91, Rub94].

Este capítulo descreve os principais conceitos do paradigma de orientação a objetos e apresenta a terminologia utilizada nesta dissertação. O capítulo é organizado da seguinte maneira: Seção 2.1 descreve os principais conceitos do modelo de objetos; Seção 2.2

apresenta uma abordagem para modelar estados através de hierarquias de classes; Seção 2.3 apresenta algumas noções sobre metodologias de análise e projeto orientados a objetos e introduz a metodologia adotada nesta dissertação; Seções 2.4, 2.5 e 2.6 apresentam os conceitos de *frameworks*, padrões de projeto e metapadrões, que são técnicas promissoras para a reutilização de software em grande escala; e, por fim, a Seção 2.7 apresenta um sumário do capítulo.

2.1 Programação Orientada a Objetos

As próximas seções descrevem os principais conceitos de programação orientada a objetos. As definições destes conceitos são baseadas nas terminologias definidas por Booch [Boo91], Rumbaugh [Rum92], Rubira [Rub94], Wolfgang Pree [Pre95], Ralph Jonhson [JF88] e Oscar Nierstrasz [Nie89].

2.1.1 Objetos

Um objeto é composto de atributos e de operações sobre esses atributos. **Atributos** são propriedades que caracterizam os estados de cada objeto; eles podem ser públicos ou privados. **Operações** caracterizam o comportamento de um objeto, e são o único meio para acessar, manipular e modificar os atributos privados de um objeto. Um objeto se comunica com outro através de **mensagens**, que identificam operações a serem realizadas no segundo objeto. Ao receber uma mensagem, um objeto executa a operação solicitada, através de computações sobre seus próprios atributos e, possivelmente, requisita a execução de métodos de outros objetos.

O conjunto de operações e atributos públicos de um objeto, que podem ser utilizados por outros objetos, formam a sua **interface pública**. Ou seja, a visão externa de um objeto é nada mais que sua interface. Em linguagens de programação orientadas a objetos, as operações que um cliente pode realizar sobre um objeto são declaradas como **métodos** na sua interface pública, que são parte da declaração da classe do objeto. A linguagem de programação C++ [Str93] usa o termo "função membro" para denotar o mesmo conceito. Para o nosso propósito, os termos método e função membro são permutáveis.

Somente através do envio de mensagens é possível solicitar a um objeto a execução de uma operação. E somente através de operações é possível mudar o estado de um objeto. Devido a estas restrições, o estado interno de um objeto é dito **encapsulado**, ou seja, o estado interno não pode ser acessado diretamente e sua representação não é visível externamente [GHJV94].

2.1.2 Classes

Uma classe descreve um conjunto de objetos com a mesma estrutura e o mesmo comportamento. Uma classe, portanto, define o conjunto de atributos e operações de suas instâncias. Um objeto é instância de uma classe.

Alguns princípios importantes de programação, tais como, abstração de dados, encapsulamento e modularidade são obtidos com o uso dos conceitos de classes e objetos. Estas características são amplamente reconhecidas como sendo boas qualidades que um software deve possuir. Portanto, uma boa aplicação do modelo de objetos favorece o desenvolvimento de software de alta qualidade.

2.1.3 Interfaces e Tipos

Os conceitos de tipos e classes são distintos. Tipo é essencialmente a descrição de uma interface. Esta interface especifica um comportamento que é comum para todos os objetos de um mesmo tipo. Uma classe especifica uma implementação particular de um tipo. Uma definição de classe, que inclui a descrição do estado interno de um objeto e seus métodos, é comum para todas as instâncias de uma classe.

O conceito de interface é fundamental em sistemas orientados a objetos. Objetos são conhecidos apenas através de suas interfaces. A interface de um objeto encapsula sua implementação — objetos de tipos diferentes são livres para implementar os métodos de forma diferente [GHJV94].

2.1.4 Herança

Herança é um mecanismo utilizado para derivar novas classes a partir de classes já existentes através de um processo de refinamento. Uma classe derivada (subclasse) herda a representação dos atributos e operações da classe base (superclasse), e pode adicionar novas operações, estender a representação de atributos e sobrepor (ou redefinir) a implementação de operações existentes. Por exemplo, na hierarquia apresentada na Figura 1b, a classe Veículo é a superclasse das classes Veículo Terrestre e Veículo Aquático.

Existem duas principais abordagens para a utilização do mecanismo de herança: hierarquia de implementação e hierarquia de tipo.

Hierarquia de Implementação consiste no uso de herança para implementar tipos de dados que são similares a outros tipos existentes. Por exemplo, supondo que já temos uma classe Lista implementada, podemos implementar uma classe Pilha como subclasse de Lista (Figura 1a). Neste caso, a operação empilhar um elemento na pilha pode ser realizada adicionando-se um elemento no final da lista e a operação desempilhar um elemento na pilha corresponde à remoção de um elemento do final da lista. É importante ressaltar que, desta maneira, o programador usa herança como uma técnica de implementação, onde não é garantido que a subclasse tem o mesmo comportamento da superclasse. O que ocorre é que uma classe já existente implementa algum comportamento que pretende-se prover também numa nova classe. Este uso do mecanismo de herança pode gerar problemas caso outras operações herdadas forneçam comportamento indesejável. Por exemplo, existem operações que são válidas sobre uma lista mas não são válidas sobre uma pilha. Portanto, herança de implementação não é recomendado, uma vez que a subclasse criada pode herdar comportamento incorreto.

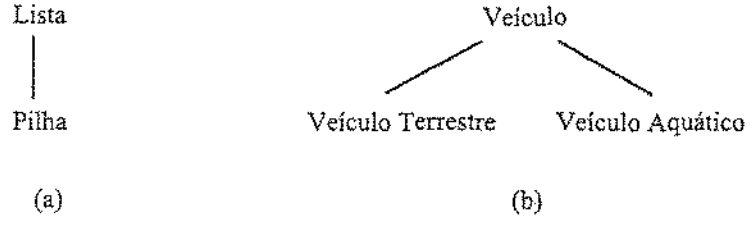


Figura 1 - Exemplos: (a) Hierarquia de Implementação e (b) Hierarquia de Tipo

Hierarquia de Tipo consiste na construção de hierarquias que garantem a relação de subtipos e supertipos. Um tipo S é um subtipo de T se S fornece pelo menos o comportamento de T. Um objeto do tipo S pode assim ser utilizado no lugar onde um objeto do tipo T é esperado, pois é garantido que ele fornece pelo menos as operações do tipo T (Princípio da Substituição). A utilização do mecanismo de herança para formar uma hierarquia de tipos relaciona o comportamento de dois tipos e não necessariamente suas implementações. Desta forma, o compartilhamento de comportamento através de herança é realizado somente quando um verdadeiro relacionamento generalização/especialização ocorre, ou seja, somente quando pode ser dito que a subclasse pode ser usada no lugar da superclasse. Quando uma classe B herda o comportamento de A, nós supomos que toda instância da classe B é uma instância da classe A porque ela se comporta da mesma maneira. A Figura 1b apresenta um exemplo de hierarquia de tipo.

Herança Múltipla

Os exemplos apresentados na Figura 1 utilizam herança simples, ou seja, cada subclasse tem exatamente uma superclasse. No caso de herança múltipla, é permitido que uma classe tenha mais de uma superclasse e, portanto, ela herda características de várias superclasses. Herança múltipla é um mecanismo poderoso para a especificação de classes, mas por outro lado, provoca a perda da simplicidade conceitual da herança simples. Na Figura 2, por exemplo, Morcego herda de Elemento Voador e Mamífero.

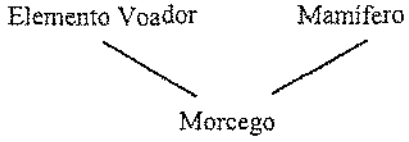


Figura 2 - Exemplo de Herança Múltipla

2.1.5 Polimorfismo

Como vimos, subclasses de uma classe A herdam métodos públicos da classe A. Desta forma, objetos que são instâncias de uma classe A1 descendente de A sempre têm pelo menos os métodos de A; os nomes de métodos e parâmetros são os mesmos, embora os

métodos possam ser redefinidos. Se a classe A1 é descendente de A, isso significa que A1 não precisa ser uma subclasse direta de A, isto é, A1 pode estar em qualquer nível da hierarquia. Como conseqüência, uma instância de uma classe descendente de A pode ser atribuída a uma variável do tipo A. Ou seja, uma variável do tipo A pode referenciar objetos da classe A, e pode referenciar também objetos de todas as classes descendentes de A. Isto implica que esta variável referencia não apenas objetos de um tipo, mas sim de "muitos tipos" — dizemos que ela é polimórfica. Portanto, podemos definir polimorfismo como sendo a habilidade de algumas variáveis referenciarem objetos de vários tipos.

2.1.6 Acoplamento dinâmico

Acoplamento dinâmico (em inglês, *dynamic binding*) implica na determinação, em tempo de execução, da implementação do método que foi chamado. Para tanto, o método a ser executado depende, em tempo de execução, do nome da mensagem e do tipo do objeto recebedor. Esta característica provê um alto grau de flexibilidade na programação.

Devido ao conceito de polimorfismo (descrito na seção anterior), uma variável a do tipo A pode fazer referência a qualquer instância que é descendente da classe A. Supondo que uma mensagem m() cujo método correspondente seja definido na classe A e, portanto, em todos os seus descendentes, seja enviado para o objeto referenciado pela variável a (em C++, a->m()). Se m() é acoplado dinamicamente, a->m() não significa necessariamente que m() seja executado com a implementação da classe A. Se o objeto referenciado por a é uma instância de um descendente de A que sobrepõe m(), a implementação que sobrepõe m() será executada.

Em C++, devemos definir explicitamente os métodos a serem acoplados dinamicamente, utilizando a palavra chave *virtual* em cada método da classe base e redefinindo-os nas classes derivadas.

2.1.7 Classes Abstratas

Uma classe abstrata é uma classe que não tem instâncias diretas, mas possui classes descendentes (chamadas de classes concretas) que têm instâncias diretas [Rum92]. Algumas operações de uma classe abstrata podem ser implementadas. Entretanto, uma

classe abstrata pode definir um protocolo para uma determinada operação sem fornecer a implementação do método correspondente. Esta operação é denominada operação abstrata. Uma operação abstrata define o protocolo de uma operação, para a qual cada subclasse concreta deve prover sua própria implementação. Como conseqüência, uma classe abstrata cria uma interface padrão para todas as classes descendentes.

A utilização de classes abstratas permite que outros componentes de software baseados nestas classes possam ser implementados sem conhecer os detalhes específicos dos objetos concretos. Para tanto, estes componentes utilizam a interface padrão das classes abstratas. Na implementação destes componentes, podem ser utilizadas referências para objetos do tipo das classes abstratas e, por meio de polimorfismo, estes componentes acessam as instâncias das classes concretas. Em C++, uma classe abstrata é criada através da declaração de pelo menos uma operação abstrata.

2.1.8 Metaclasses

Metaclasse é uma classe cuja instância é um objeto que representa uma classe. As informações comuns a todas as instâncias da classe podem ser armazenadas em atributos da metaclasse. Estes atributos são bastante úteis para armazenar informações predefinidas para a criação de novos objetos ou informações sobre as instâncias das classes. Métodos associados à metaclasse podem ser utilizados para recuperar e atualizar os valores dos atributos da metaclasse. O método mais comum de uma metaclasse é o método para a criação de instâncias de classes. Cada metaclasse pode ter, por exemplo, seu próprio método *new* para a criação e iniciação das instâncias da classe.

2.1.9 Delegação

Delegação [JZ91, Rub94] é um mecanismo semelhante ao de herança, utilizado para compartilhar comportamento em sistemas orientados a objetos. Herança aplica-se a classes e delegação aplica-se a objetos. Em princípio, a relação de delegação pode ser estabelecida dinamicamente, enquanto que a relação de herança é estabelecida quando a classe é criada.

Em delegação, um objeto que recebe uma mensagem pode delegar sua execução para seu objeto delegado [GHJV94]. Por exemplo, ao invés da criação de uma classe Janela

derivada de uma classe Retângulo, a classe Janela pode reutilizar o comportamento da classe Retângulo, declarando uma referência para um objeto Retângulo e delegando operações para este objeto. Um objeto Janela, portanto, envia mensagens para sua instância de Retângulo, ao invés de herdar as operações. A Figura 3 mostra um exemplo, onde Janela delega sua operação Área para Retângulo.

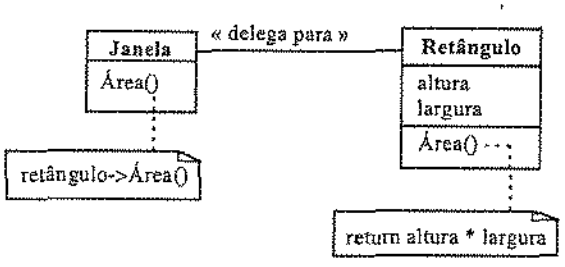


Figura 3 - Exemplo do Mecanismo de Delegação

A principal vantagem do mecanismo de delegação é facilitar o compartilhamento de comportamentos em tempo de execução. No exemplo da Figura 3, um objeto do tipo Janela pode tornar-se circular em tempo de execução, simplesmente substituindo a instância de Retângulo por uma instância de Círculo, supondo que Retângulo e Círculo tenham o mesmo supertipo [GHJV94].

Embora herança e delegação sejam geralmente definidas como alternativas para o projeto de sistemas orientados a objetos, os dois mecanismos podem ser usados em linguagens baseadas em classes como C++.

Delegação versus Herança

A principal vantagem que delegação tem sobre herança é que delegação facilita objetos mudarem a implementação de seus comportamentos [JZ91]. Um objeto delegado não faz nenhuma suposição sobre a representação de seu delegante. A maioria das linguagens de programação orientadas a objetos não permitem mudar a classe de um objeto. Entretanto, é bastante fácil mudar o delegado de um objeto. Além disso, uma linguagem com verificação estática de tipos, como C++, pode assegurar que um delegado sempre entende todas as mensagens delegadas para ele.

Delegação versus Envio de Mensagem

É comum um objeto ter que colaborar com outro objeto para realizar uma determinada tarefa. Para tanto, um objeto pode enviar uma mensagem ao outro objeto solicitando a execução de uma operação. Porém, o mecanismo de delegação é mais do que apenas enviar uma mensagem a outro objeto. Delegação é poderoso o suficiente para simular herança, ao passo que um simples envio de mensagem não simula a propriedade *self* — Em C++, *this* (Exemplo na Figura 4). Quando um objeto delegado envia uma mensagem para *self*, ela deve ser localizada no objeto delegante. Em outras palavras, mensagens que o delegado envia para si mesmo são recebidas pelo delegante. Portanto, delegação difere de envio de mensagem, pois o delegante continua fazendo o papel de destinatário, mesmo depois de delegar a mensagem. Para tanto, o mecanismo de delegação é implementado em linguagens como C++ incluindo o destinatário original como um argumento extra para cada mensagem delegada. Na Figura 5, por exemplo, um objeto da classe Carro delega o método Move para um objeto da classe Veículo, incluindo o destinatário original (*this*) como um argumento para a mensagem delegada.

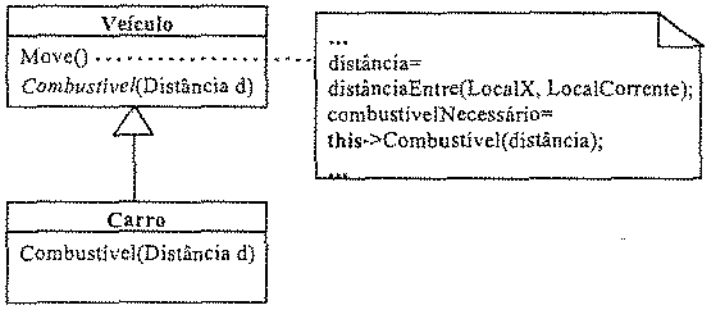


Figura 4 - Exemplo de utilização da Propriedade *Self* (Em C++, *this*)

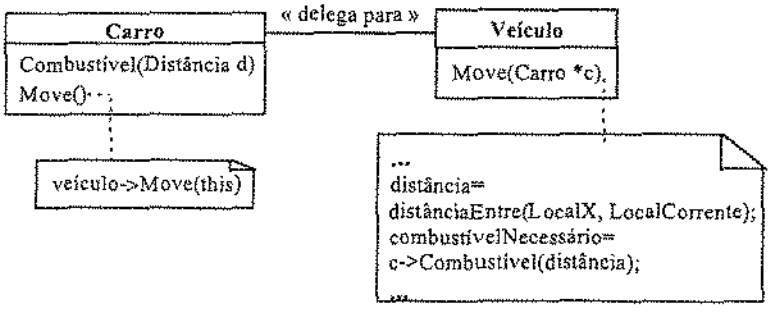


Figura 5 - Simulação da Propriedade *Self* utilizando delegação

2.2 Hierarquia de Estados e Delegação

Como vimos, uma classe descreve um conjunto de objetos com a mesma estrutura e o mesmo comportamento. Muita vezes, entidades do mundo real exibem diferentes fases de comportamento durante o seu tempo de vida. O termo **estado** é utilizado para designar cada uma destas fases de comportamento. Em um determinado momento, instâncias diferentes de uma classe podem se encontrar em diferentes estados. O estado em que uma instância se encontra é denominado **estado corrente** [Rub94].

Em geral, informações sobre estados são obtidas durante a fase de projeto do sistema. Com isso, quase sempre estas informações são implementadas dentro dos métodos dos objetos. Porém, existem outras alternativas de projeto que podem representar esses estados mais explicitamente. A seguir, apresentamos um exemplo para ilustrar estas idéias. O exemplo consiste em um simples esquema de paginação em memória virtual (extraído de [SC95]). Uma página de memória virtual pode ser mapeada em memória física, tornando-se acessível (página ativa). Posteriormente, esta página pode ser armazenada de volta, liberando a memória física para outros usos (página inativa).

Solução 1

Utilizando o conceito de herança e de acoplamento dinâmico, podemos criar uma hierarquia de classes, incluindo a descrição dos diferentes estados de uma página, como mostra a Figura 6 (em notação UML, descrita na Seção 2.3.1). Nesta solução, os métodos LiberaPágina e AcessaPágina são definidos como *virtuais* e sobrepostos nas subclasses PáginaATIVA e PáginaINATIVA. Desta forma, a implementação destes métodos contêm o código específico das respectivas operações.

Esta solução captura o relacionamento entre os dois estados de uma página, porém ela tem pelo menos duas limitações. Primeiro, linguagens de programação baseadas em classes, como C++, não permitem que um programa mude a classe de seus objetos. Quando uma página muda de estado, por exemplo, de PáginaATIVA para PáginaINATIVA, um novo objeto PáginaINATIVA deve ser criado e o objeto antigo PáginaATIVA deve ser eliminado. Essa operação de eliminação e criação de objetos pode causar uma grande sobrecarga no tempo de execução para aplicações com um tamanho razoável. Além disso, não é possível

ao programador garantir que a identidade do novo objeto seja idêntica à identidade do objeto eliminado. Surge assim um problema crucial, se outros objetos do sistema mantiverem referências para o objeto antigo. Uma segunda limitação é que PáginaATIVA e PáginaINATIVA não são realmente classes subtipos/especializações de uma classe abstrata Página, mas sim conceitos de estados diferentes, de uma mesma abstração. Os estados lógicos de uma página são classificados como ativo e inativo e não a própria página.

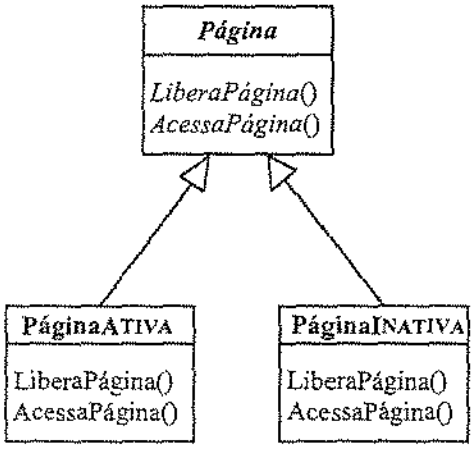


Figura 6 - Exemplo de Representação de Estados usando Herança

Solução 2

Dois diferentes contextos foram misturados na primeira solução proposta: a classe de um objeto Página e seu estado corrente lógico. Esses dois contextos estão representados na mesma hierarquia de classes (Figura 6). Entretanto, podemos separar esses contextos introduzindo uma hierarquia de classes para representar exclusivamente os diferentes estados do objeto página. Neste caso, temos a classe Página, que oferece a interface para as operações com páginas e temos a hierarquia que representa os diferentes estados lógicos que um objeto Página pode desenvolver durante o seu tempo de vida (Figura 7).

Um objeto Página delega seu comportamento para o objeto EstadoPágina: um objeto "interno" que muda dinamicamente de estado (PáginaATIVA e PáginaINATIVA), de acordo com as mudanças no estado lógico do objeto Página. Para tanto, a classe Página mantém um apontador para um objeto estado (uma instância de uma subclasse de EstadoPágina), que representa o estado corrente da Página. A operação LiberaPágina, por exemplo, é

enviada, em tempo de execução, para o objeto que representa o estado corrente (apontador estado na Figura 7). Nesta solução, o estado lógico de um objeto Página pode mudar sem a necessidade de excluir e recriar o objeto, ou seja, quando uma página muda de estado, o apontador para o estado corrente aponta para a instância da subclasse que implementa o novo estado.

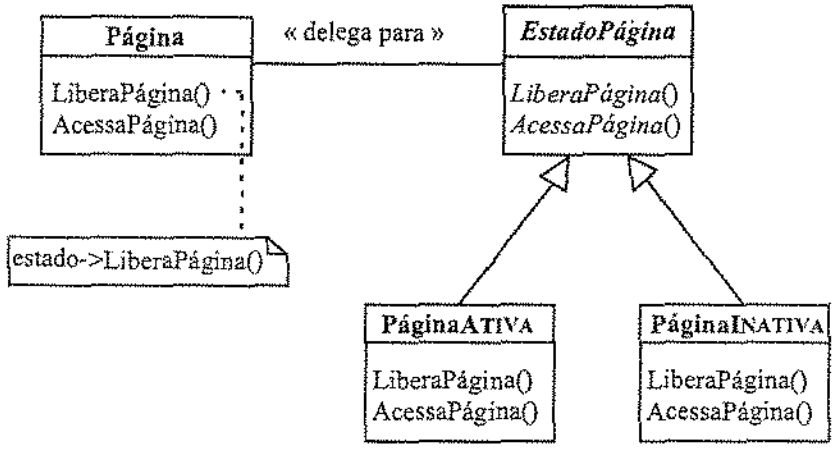


Figura 7 - Exemplo de Hierarquia de Estados e Delegação

A especificação da interface da hierarquia de estados é definida pela classe base abstrata EstadoPágina, que declara as operações abstratas LiberaPágina e AcessaPágina.

2.3 Metodologia de Análise e Projeto

Uma metodologia é um conjunto de regras e notações utilizados para construir um modelo abstrato de um determinado problema. À medida que a complexidade de um sistema aumenta, a construção de um modelo abstrato torna-se indispensável. Desta forma, a utilização de uma boa metodologia torna-se um fator essencial.

Várias metodologias orientadas a objetos podem ser encontradas na literatura [Boo91, Rum92, Jac92]. Existem um grande número de metodologias com diferentes notações e regras muitas vezes conflitantes. De fato, não existe uma representação padrão para os conceitos de orientação a objetos nessas diferentes metodologias. Entretanto, esforços de padronização têm proposto uma nova linguagem para modelagem de sistemas — a UML (*Unified Modeling Language*) [BRJ97].

A UML representa a unificação das metodologias de Booch [Boo91], OMT [Rum92, Rum95a, Rum95b] e OOSE [Jac92]. Esta unificação teve como principais objetivos: eliminar os elementos que não eram utilizados na prática, adicionar elementos eficazes de outros métodos e criar novos elementos para soluções não disponíveis. O resultado é a proposta de uma linguagem de modelagem simples, porém bem mais expressiva e mais uniforme do que as metodologias que formaram a base dessa unificação.

Diante destas perspectivas, adotamos a UML como metodologia para construir o modelo orientado a objetos do nosso experimento. A próxima Seção faz uma breve descrição das características desta metodologia.

2.3.1 Metodologia UML

A UML é uma linguagem para especificação, construção, visualização e documentação dos artefatos[1] de um sistema de computação [BRJ97]. Ela pretende ser uma linguagem universal para modelagem de sistemas, isto é, pretende expressar modelos de vários tipos de aplicações com propósitos variados. A existência de um processo[2] universal simples para todos os estilos de desenvolvimento não parece possível: o que funciona para um projeto compacto de software provavelmente não serve para um tipo de sistema de grande porte, distribuído e crítico para o ser humano. Entretanto, a UML pode ser usada para expressar os artefatos de todos estes diferentes processos através dos modelos que são produzidos. Ela destina-se, inclusive, para modelagem de sistemas de tempo real, e sistemas distribuídos e concorrentes.

A UML supõe que o processo é orientado a casos de usos, centrado numa determinada arquitetura, iterativo e incremental. Os detalhes do processo geral de desenvolvimento devem ser adaptados para a cultura particular de desenvolvimento ou para o domínio da aplicação de uma organização específica. Esta separação entre a linguagem de modelagem e os processos possibilita aos usuários da metodologia um grau considerável de liberdade para desenvolver um processo específico usando uma linguagem comum de expressão.

---

[1] Artefatos englobam a arquitetura, ou seja, os conceitos e modelos que apoiam o desenvolvimento de uma estrutura; bem como os métodos, que descrevem como trabalhar com estes conceitos e modelos para um desenvolvimento ideal [Jac92].

[2] O processo é o desenvolvimento completo de um produto, durante todo o seu ciclo de vida [Jac92].

Os principais recursos que os desenvolvedores utilizam para manipular modelos são os diagramas. Um diagrama é a projeção dos elementos de um modelo; os mesmos elementos podem aparecer em múltiplos diagramas. Um diagrama pode apresentar alguns — mas não necessariamente todos — detalhes de um elemento particular. A UML define um conjunto central de diagramas, os quais provêem múltiplas perspectivas de um sistema que se encontra sob análise ou desenvolvimento:

- Diagrama de Casos de Uso, que é utilizado para delimitar o sistema e definir a funcionalidade que ele pode oferecer;
- Diagramas da Estrutura Estática:
    - Diagrama de Classes, que mostra a estrutura estática do sistema através da descrição de seus componentes (classes, objetos, módulos, composições, etc) e seus relacionamentos;
    - Diagrama de Objetos, que representa uma instância de um diagrama de classes; ele mostra um quadro do estado de um sistema em determinado momento;
- Diagramas de Comportamento:
    - Diagrama de Estados, que descreve a evolução temporal (seqüência de estados) de um objeto de uma determinada classe, em resposta às interações com outros objetos internos ou externos ao sistema.
    - Diagrama de Seqüências, que é utilizado para modelar cenários, ilustrando as principais interações entre objetos.
    - Diagrama de Colaborações, que mostra a seqüência de mensagens que implementa uma operação ou uma transação envolvendo um conjunto de objetos.
- Diagramas de Implementação:
    - Diagrama de Componentes, que descreve o projeto físico de um sistema, ou seja, os componentes de software e hardware que implementam o projeto lógico.
    - Diagrama de Disposições, que especifica a topologia física (plataforma) na qual um sistema executa.

Uma descrição detalhada desse conjunto de diagramas encontra-se em [BRJ97]. Não temos a intenção de descrevê-los nesta dissertação; apenas comentamos, na Seção 2.3.2.1, a notação gráfica dos principais componentes do diagrama de classes. Antes, porém, enumeramos alguns dos novos conceitos incluídos na UML.

2.3.1.1 Novos Conceitos Incluídos na UML

Como mencionado, a UML, além de unificar as metodologias, também criou novos elementos para soluções até então não disponíveis, tais como:

- estereótipo - um novo tipo de elemento de modelagem que estende a semântica da UML. Certos estereótipos são predefinidos na UML, outros podem ser definidos pelo usuário;
- restrição - semântica para condição ou restrição. Certas restrições são predefinidas na UML, outras podem ser definidas pelo usuário;
- *threads* e processos;
- distribuição e concorrência;
- padrões/colaborações;
- clara distinção entre tipo, classe e instância;
- refinamento - relacionamento entre duas descrições de uma mesma coisa entre diferentes níveis de abstração. A evolução de um projeto pode ser descrita através de relacionamentos de refinamento;
- componentes e interfaces entre componentes;

2.3.1.2 Diagrama de Classes

O diagrama de classes é o núcleo da UML. Ele mostra a estrutura estática do sistema: os conteúdos e seus relacionamentos. Esta seção apresenta a notação gráfica dos principais componentes deste diagrama.

Classes

A Figura 8 apresenta a notação para classes. Uma classe é representada por uma caixa retangular de linhas sólidas, com três compartimentos. Os três compartimentos contêm, de

cima para baixo: o nome da classe, uma lista de atributos (com tipos e valores iniciais opcionais) e uma lista de operações (com lista de parâmetros e tipos de retorno opcionais). Os nomes de classes ou operações abstratas são descritos em itálico.

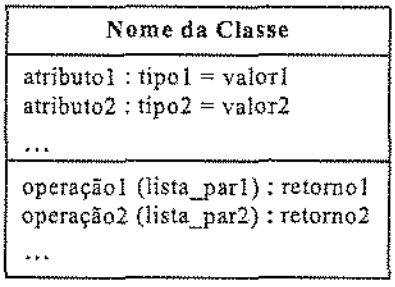


Figura 8 - Classe

Objetos

Objetos são representados por um retângulo, semelhante ao das classes. Para diferenciá-los, o nome do objeto no retângulo é sublinhado. O nome da classe do objeto pode opcionalmente suceder o nome do objeto, separados pelo símbolo ":" (Figura 9). Um objeto pode apresentar também um segundo compartimento para mostrar os atributos e seus valores.

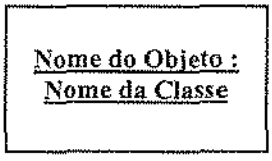


Figura 9 - Objeto

Estereótipos

Um estereótipo representa a metaclassificação de um elemento. Desta forma, permite aos usuários adicionar novos recursos de modelagem, que não fazem parte do núcleo da UML. Estereótipos têm implicações semânticas que podem ser específicas para cada tipo de estereótipo.

A sintaxe para estereótipos é representada por um texto normal entre os símbolos "«" e "»", como mostra a Figura 10. Nesta figura, temos uma classe denominada CálculoInválido,

cujo estereótipo é exceção. Outro estereótipo é definido para a operação LerValor, que é classificada como operação de acesso.

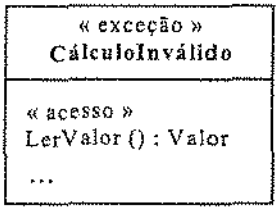


Figura 10 - Exemplo da Sintaxe de Estereótipos

Relacionamentos

Uma ligação é uma conexão estrutural entre objetos de diferentes classes. Um relacionamento descreve um grupo de ligações com semânticas e estruturas comuns. Em outras palavras, um relacionamento descreve um conjunto de potenciais ligações, assim como uma classe descreve um conjunto de potenciais objetos. A Figura 11 mostra os tipos de relacionamento da UML.

Uma **associação** significa que uma classe cliente usa algumas facilidades de uma classe servidor. Os tempos de vida dos objetos que participam deste relacionamento são independentes. Uma associação pode ter um nome, bem como uma indicação de direção (um triângulo sólido sem cauda).

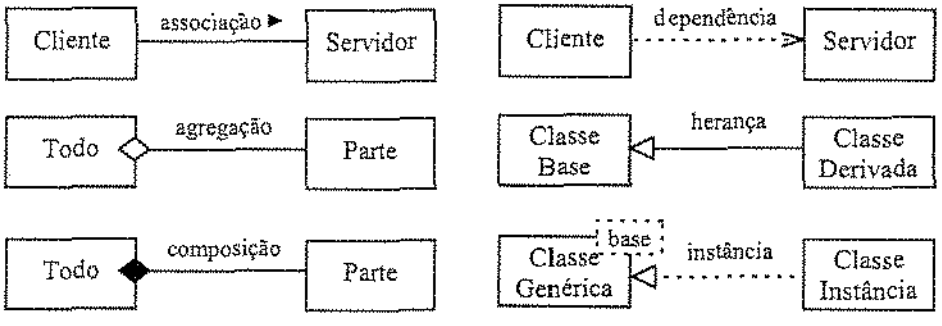


Figura 11 - Tipos de Relacionamentos

Uma **agregação** tem a conotação de relacionamento "uma parte de", onde objetos representando componentes são associados com um objeto representando o todo. Uma

agregação é representada como uma associação, acrescentando o desenho de um "diamante vazio" no objeto que representa o todo no relacionamento.

Uma **composição** é uma forma estendida de agregação, onde os relacionamentos entre as partes são válidos somente dentro da composição. Portanto, o tempo de vida dos componentes dependem do tempo de vida do todo. Uma composição é representada como uma agregação, porém com o desenho do "diamante cheio". Alternativamente, a UML provê uma representação mais conveniente para composição, que consiste em aninhar graficamente os componentes dentro do todo.

Uma **generalização** (herança) é o relacionamento entre uma classe e uma ou mais versões refinadas dela. A classe sendo refinada é chamada superclasse (ou classe base) e cada versão refinada é chamada subclasse (ou classe derivada). Assim, diz-se que cada subclasse herda as características de sua superclasse. O relacionamento de generalização é representado por uma linha direcionada, da subclasse para a superclasse, com uma seta triangular sem preenchimento na superclasse. Por convenções de leiaute, as linhas de várias subclasses podem ser desenhadas como uma árvore compartilhando uma linha simples para a seta triangular. No caso de herança múltipla, simplesmente desenhamos múltiplos relacionamentos de generalização a partir da subclasse.

Uma **dependência** significa que uma classe depende de algum serviço de uma outra classe, mas não tem uma referência interna ou ponteiro para a mesma.

Um relacionamento **instância** ocorre entre uma classe genérica (*template*) e a classe que resulta da criação da instância.

Uma classe que participa de uma relacionamento pode especificar a sua multiplicidade, isto é, quantas instâncias da classe podem ser associadas a uma instância de outra classe. Multiplicidade é indicada por uma expressão (intervalo de inteiros). Um intervalo é indicado por um inteiro (menor valor), dois pontos (..), e um inteiro (maior valor); um inteiro simples é um intervalo válido, e o símbolo "*" indica "muitos", isto é, um número ilimitado de objetos. O símbolo "*" sozinho equivale ao intervalo "0..*", isto é, qualquer número inclusive nenhum. Uma função escalar opcional tem a multiplicidade "0..1". Se a multiplicidade é maior do que um, a palavra chave {ordenado} pode ser colocada na

função, indicando que os elementos têm uma determinada ordem; caso contrário, eles são um conjunto não ordenado.

Uma classe pode estar relacionada a si mesma. Podemos também ter múltiplos relacionamentos entre um par de classes. Neste caso, é importante utilizar nomes nos relacionamentos.

2.3.1.3 Módulos

Modelos grandes requerem organização interna. Um módulo (*package*) é um subconjunto de um modelo. Todos os elementos e diagramas da UML podem ser organizados em módulos. Módulos são puramente organizacionais; eles não adicionam semântica ao modelo. Nós os representamos como uma "pasta com aba".

A Figura 12 mostra o exemplo de um módulo (Módulo A) parcialmente expandido, que possui outros módulos (X, Y e Z). Um modelo do mundo real normalmente é mapeado em vários módulos.

Módulos representam um mecanismo interessante para organizar modelos. Eles podem ser usados não somente para designar grupamentos lógicos, mas também para designar grupos de casos de uso e grupos de processadores. Em cada caso, a semântica usual de módulo se aplica. Pode-se também utilizar estereótipos para distinguir um tipo de módulo de outro.

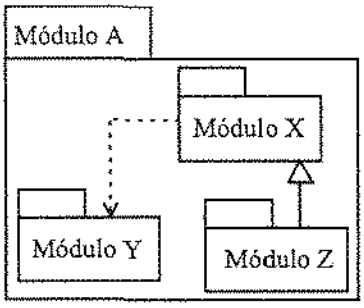


Figura 12 - Exemplo de Módulos e suas Dependências

2.4 *Frameworks*

Uma das principais vantagens do paradigma de orientação a objetos é o seu suporte para reutilização de software [WJ90]. Entretanto, tem sido observado que a reutilização de classes ou mesmo de bibliotecas de classes (baseadas em herança) não provê um grau satisfatório de reutilização. Classes e objetos possuem granularidade pequena para alcançar o nível de reutilização desejado [Fir93]. O conceito de *frameworks* orientados a objetos tem sido proposto como alternativa para alcançar reutilização de software em grande escala.

Um *framework* [JF88, WJ90] é um conjunto de classes (abstratas e concretas), que provê uma infra-estrutura genérica de soluções para um conjunto de problemas. Ao contrário das abordagens tradicionais para reutilização de software, que consistem em construir bibliotecas de classes, *frameworks* permitem reutilizar não apenas componentes isolados mas toda a arquitetura de um domínio específico. Em outras palavras, um *framework* provê um projeto genérico que pode ser adaptado segundo as necessidades de cada aplicação específica. Este projeto corresponde à especificação de um conjunto de classes, suas interfaces e o modo como elas se relacionam.

A Figura 13 ilustra as diferenças entre *frameworks* e bibliotecas de classes convencionais. *Frameworks* exibem uma inversão de controle em tempo de execução. Aplicações baseadas em bibliotecas de classes colocam o controle (fluxo de eventos) na própria aplicação. O programador da aplicação é encarregado de projetar/implementar o fluxo de controle da aplicação específica. Em contrapartida, aplicações baseadas em *frameworks* colocam o fluxo de controle no próprio *framework*. Programadores que utilizam um *framework* implementam código que será chamado pelo *framework* (*"callbacks"*), ao invés de simplesmente implementarem código que invoca bibliotecas de classes. Desta forma, aplicações reutilizam o fluxo de controle e a arquitetura de software que o *framework* provê [LK94]. Enquanto os componentes de uma biblioteca de classes são utilizados individualmente, classes em um *framework* são reutilizadas como um todo, para resolver uma instância específica de um certo problema.

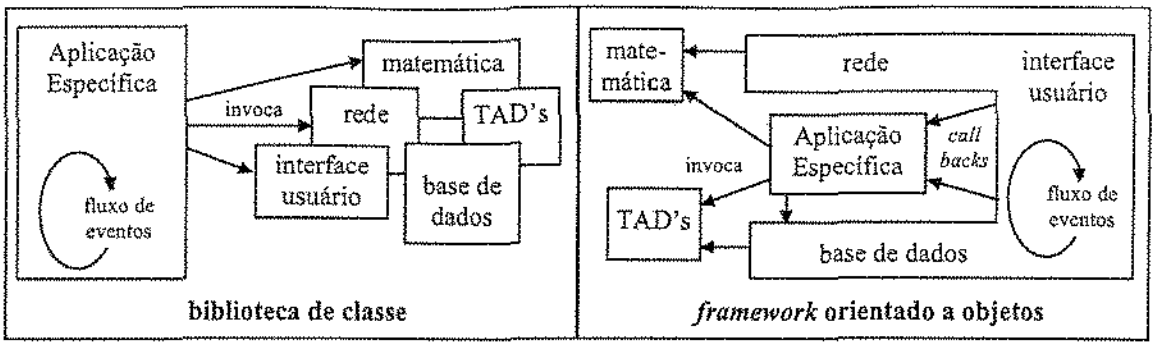


Figura 13 - Diferenças entre Biblioteca de Classes e *Frameworks*

A utilização de um *framework* compreende tipicamente as seguintes atividades:

- definição de novas classes (subclasses) necessárias para a aplicação específica;
- sobreposição de métodos de algumas classes do *framework* em subclasses;
- configuração de um conjunto de objetos, provendo parâmetros para cada objeto e conectando-os;

Em geral, um *framework* antecipa a maior parte do projeto de um sistema. Mesmo quando novas classes são necessárias, a maior parte do trabalho de utilização de um *framework* consiste em adaptar e configurar objetos.

O projeto de um *framework* bem estruturado é mais do a extração das classes abstratas de um sistema. Um sistema tem que funcionar apenas para uma aplicação, mas um *framework* tem que funcionar para várias aplicações correlatas. Assim, um *framework* é uma generalização de um grupo de aplicações que podem ser construídas a partir dele [WJ90]. No Capítulo 5 apresentamos o projeto de um *framework* para o domínio específico de controladores de software para trens.

2.5 Padrões de Projeto

Padrões de projeto (em inglês, *Design Patterns*) [GHJV94, Hel95, RZ96] têm sido propostos como meio de representar, registrar e reutilizar micro-arquiteturas de projeto repetitivas, bem como a experiência acumulada pelo projetista ao desenvolver estas estruturas. Um padrão nomeia, abstrai e identifica os aspectos chaves das estruturas de projetos, identificando classes e instâncias, suas colaborações e a distribuição de responsabilidades. Eles formam uma base de experiência para construir sistemas

reutilizáveis, atuando como blocos pré-fabricados para a construção de sistemas mais complexos.

Em [GHJV94] é apresentado um catálogo contendo uma série de padrões de projeto orientados a objetos, independentes de aplicações e linguagens de programação. Estes padrões representam descrições de objetos e classes que são ajustados para resolver um problema de projeto geral em um contexto particular. O catálogo descreve os padrões através de quatro elementos essenciais:

- O **nome do padrão** é um identificador que pode ser utilizado para sintetizar um problema de projeto, suas soluções e conseqüências, através de uma ou duas palavras. O nome de um padrão aumenta o vocabulário de projeto e também o nível de abstração do mesmo;
- O **problema** descreve quando o padrão deve ser aplicado. Ele explica o problema e seu contexto.
- A **solução** descreve os elementos (classes e objetos) que constituem o projeto, seus relacionamentos, responsabilidades e colaborações. A solução não descreve um projeto ou implementação particular, porque um padrão é uma espécie de modelo que pode ser aplicado em várias situações diferentes;
- As **conseqüências** são os resultados (vantagens e desvantagens) e as decisões que decorrem da utilização do padrão. A descrição destas conseqüências é vital para uma avaliação dos custos e benefícios de um padrão;

A Figura 14 mostra um exemplo simplificado de padrão de projeto encontrado no catálogo de padrões [GHJV94, Página 293]. Existem situações de projetos em que é preciso definir uma dependência (um para muitos) entre objetos de forma que, quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente. O padrão Observador (Figura 14) descreve como estabelecer esses relacionamentos. Os objetos chave deste padrão são os objetos Observado e Observador. Um observado pode ter qualquer número de observadores. Todos os observadores são notificados quando o observado realiza uma mudança de estado. Com isso, cada observador consulta o observado para sincronizar seu estado.

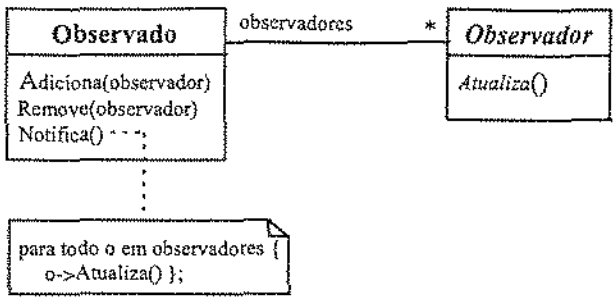


Figura 14 - Exemplo de Padrão de Projeto (Padrão Observador)

É comum a associação entre os conceitos de padrões e *frameworks*. Padrões podem ser vistos como descrições abstratas de *frameworks*, que facilitam reutilização de arquiteturas de software. *Frameworks* podem ser vistos como realizações concretas de padrões que facilitam a reutilização direta de projeto e código. Uma diferença clara entre padrões e *frameworks* é que padrões são descritos de uma maneira independente de linguagens de programação, ao passo que *frameworks* são geralmente implementados em uma linguagem particular [Sdt95b]. Segundo R. Johnson [Joh92], a próxima geração de *frameworks* embutirá explicitamente dúzias ou centenas de padrões — e padrões serão bastante utilizados para documentar a forma e o conteúdo de *frameworks*.

2.6 Metapadrões

Metapadrões (em inglês, *Metapatterns*) representam uma abordagem proposta por Wolfgang Pree [Pre95], que consiste na especificação de um conjunto de padrões que descrevem como construir *frameworks* independente de um domínio específico. Segundo Pree, " (...) metapadrões constituem uma abordagem elegante e poderosa que pode ser aplicada para classificar e descrever padrões de projeto. Portanto, metapadrões não substituem as abordagens de padrões de projeto mas complementam-as (...)".

Como mencionado anteriormente, uma das atividades realizadas quando utilizamos um *framework* consiste em definir subclasses a partir das classes do *framework* e sobrepor métodos destas classes nas subclasses. A abordagem por metapadrões distingue dois tipos principais de métodos para a implementação de um *framework*: método genérico (em inglês, *template method*) e método componente (em inglês, *hook method*). Estes métodos formam os metapadrões requeridos para projetar *frameworks* consistindo de classes simples

ou grupos de classes e suas interações. De um modo geral, os métodos genéricos implementam a parte fixa do *framework* e os métodos componentes implementam as partes adaptáveis do *framework*. Podemos dizer que métodos complexos denominados métodos genéricos podem ser implementados baseados em métodos elementares denominados métodos componentes.

O conceito de métodos genéricos e métodos componentes é ilustrado na Figura 15. Chamadas de métodos são expressas por setas. O método genérico M1() chama um método abstrato M2() e um método comum M3(). M2() consiste em um ponto adaptável (método componente) que deve ser preenchido e M3() consiste em um ponto adaptável (método componente) que pode ser substituído.

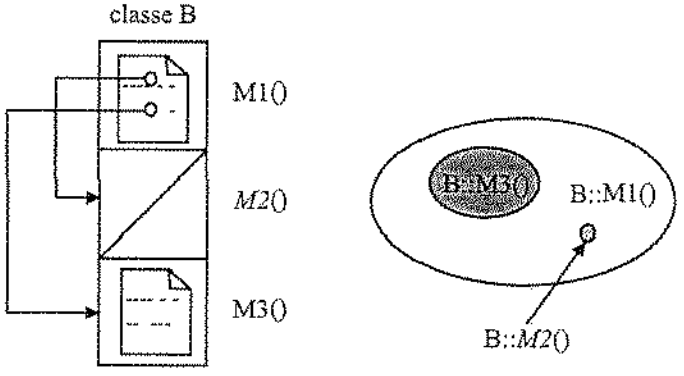


Figura 15 - Método Genérico (M1) invocando seus Métodos Componentes (M2 e M3)

A classe B (Figura 15) pode, por exemplo, representar uma classe Item Alugado, que poderia fazer parte de um *framework* para sistemas de locação (Figura 16). O método genérico M1() da classe B corresponde ao método ImprimeFatura da classe Item Alugado. O método CalculaDiária é um método componente, que define somente a sua interface e deve ser preenchido (de acordo com um cálculo específico de um determinado item alugado como, por exemplo, um automóvel) e o método RetornaNomeItem é um método componente que provê uma implementação predefinida, mas pode ser substituído (podemos considerar, por exemplo, que o item alugado é por padrão, um quarto de hotel).

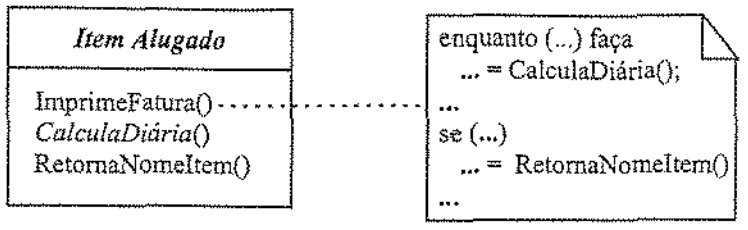


Figura 16 - Exemplo de Método Genérico e Métodos Componentes

Quando métodos genéricos são unificados em uma única classe, o comportamento desta classe pode ser modificado somente através da definição de uma subclasse. Entretanto, em algumas situações é necessário maior flexibilidade para permitir adaptações em tempo de execução. Para obter-se este grau de flexibilidade, métodos genéricos e métodos componentes podem ser declarados em classes separadas. Na Figura 17, por exemplo, o método genérico AtivaItemDisponível tem como método componente, o método ImprimeFatura, declarado na classe Item Alugado.

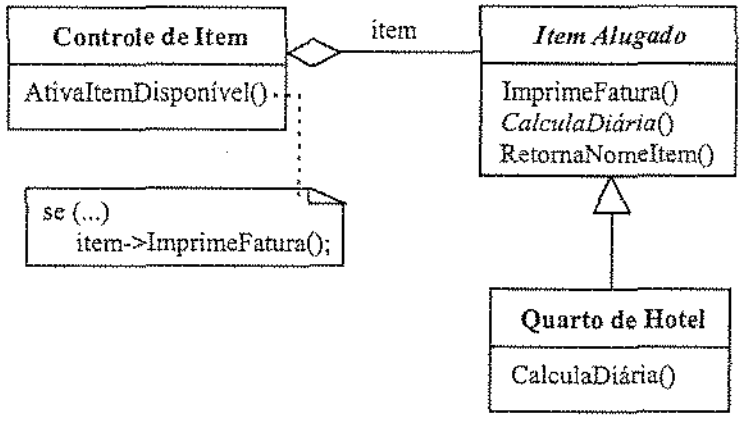


Figura 17 - Método Genérico e Método Componente Declarados em Classes Separadas

Pree define um conjunto de sete metapadrões que implementam o comportamento dos métodos componentes e genéricos. A classe que implementa os métodos componentes é denominada classe componente (H), ao passo que a classe que implementa os métodos genéricos é denominada classe genérica (T). Em outras palavras, uma classe componente parametriza uma classe genérica [Pre95].

Um caso especial ocorre quando a classe genérica e a classe componente são unificadas em uma única classe, resultando no **metapadrão Unificação**. A Figura 18

apresenta o diagrama de classe deste metapadrão. No exemplo apresentado na Figura 16, temos a ocorrência do mesmo.

Figura 18 - Metapadrão Unificação

No **metapadrão Conexão 1:1**, um objeto de uma classe genérica refere-se a exatamente um objeto da classe componente. Não existe relacionamento de herança entre a classe genérica e a classe componente. A Figura 19a apresenta o diagrama de classe deste metapadrão. No exemplo apresentado na Figura 17, temos a ocorrência do mesmo.

No **metapadrão Conexão 1:N** (Figura 19b), um objeto de uma classe genérica refere-se a qualquer número de objetos da classe componente.

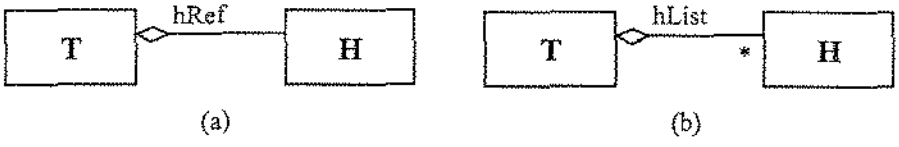


Figura 19 - Metapadrões (a) Conexão 1:1 e (b) Conexão 1:N

No **metapadrão Conexão Recursiva 1:1**, um objeto de uma classe genérica refere-se a exatamente um objeto de sua classe componente. A classe genérica é descendente da classe componente (Figura 20a). Em uma versão modificada deste metapadrão, classes genéricas e classes componentes são unificadas em uma única classe, resultando no **metapadrão Unificação Recursiva 1:1** (Figura 20b).

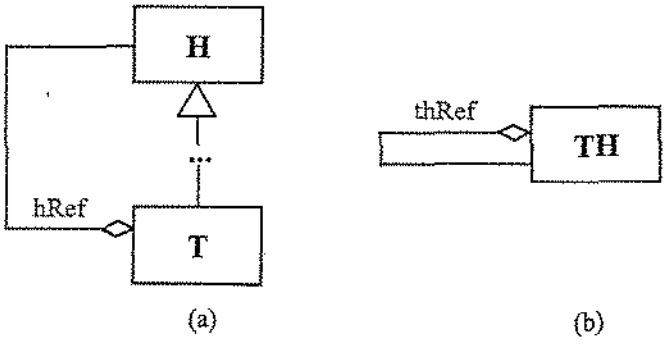


Figura 20 - Metapadrões (a) Conexão Recursiva 1:1 e (b) Unificação Recursiva 1:1

No **metapadrão Conexão Recursiva 1:N**, um objeto de uma classe genérica refere-se a qualquer número de objetos de sua classe componente. A classe genérica é descendente da classe componente (Figura 21a). Em uma versão modificada deste metapadrão, classes genéricas e classes componentes são unificadas em uma única classe, resultando no **metapadrão Unificação Recursiva 1:N** (Figura 21b).

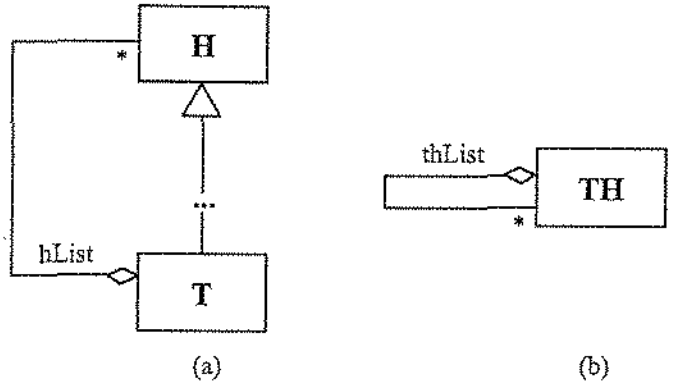


Figura 21 - Metapadrões (a) Conexão Recursiva 1:N e (b) Unificação Recursiva 1:N

Na Figura 22 temos um exemplo da aplicação do metapadrão Conexão Recursiva 1:N. Neste exemplo, o método genérico TamanhoEmBytes da classe Diretório é implementado invocando-se o método componente TamanhoEmBytes da classe Item do Diretório.

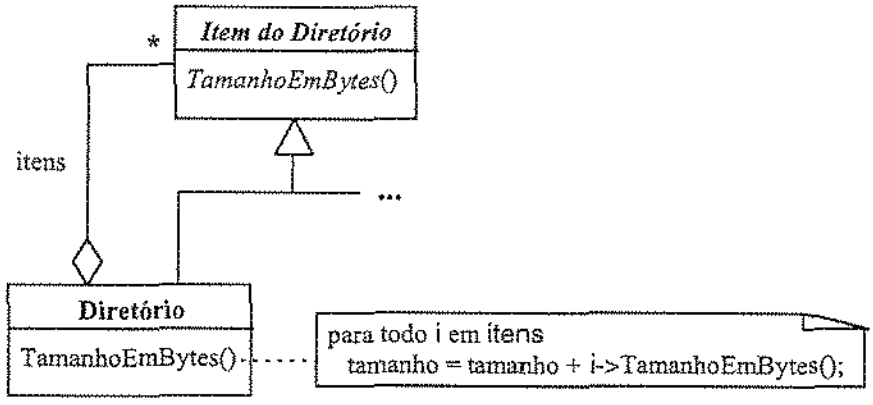


Figura 22 - Aplicação do Metapadrão Conexão Recursiva 1:N

2.7 Sumário

O paradigma de orientação a objetos descreve um estilo de programação baseado nas noções de classes, objetos e herança. Uma classe descreve um conjunto de objetos com a mesma estrutura e o mesmo comportamento. O comportamento de um objeto é caracterizado pela sua interface, ou seja, o conjunto de operações públicas da sua classe. Um objeto é composto de atributos e de operações que são realizadas sobre esses atributos. Ao receber uma mensagem, o objeto executa a operação solicitada, através de computações sobre seus próprios atributos e, possivelmente, requisita a execução de operações de outros objetos. Herança é um mecanismo para derivar novas classes a partir de classes já existentes através de um processo de refinamento. Uma classe derivada herda a representação dos atributos e operações da classe base, mas pode adicionar novas operações, estender a representação de atributos ou sobrepor a implementação de operações já herdadas. Deste modo, a programação orientada a objetos consiste em definir classes e criar hierarquias de classes, onde as propriedades comuns podem ser transmitidas das superclasses para as subclasses através do mecanismo de herança.

Outros conceitos importantes do paradigma de orientação a objetos são: polimorfismo, acoplamento dinâmico, classes abstratas e delegação. **Polimorfismo** pode ser definido como sendo a habilidade de algumas variáveis referenciarem objetos de vários tipos. **Acoplamento dinâmico** implica na determinação, em tempo de execução, da implementação do método que foi chamado. Para tanto, o método a ser executado depende, em tempo de execução, do nome da mensagem e do tipo do objeto recebedor. Esta característica provê um alto grau de flexibilidade na programação. **Classes abstratas** são classes que definem um comportamento comum, que deve ser implementado por subclasses concretas. Portanto, uma classe abstrata não tem instâncias diretas, mas possui classes descendentes que têm instâncias diretas. Classes abstratas permitem definir interfaces padrões, que podem ser utilizadas por outros componentes de software, de maneira transparente. **Delegação** [JZ91, Rub94] é um mecanismo semelhante ao de herança, utilizado para compartilhar comportamento em sistemas orientados a objetos. Herança aplica-se a classes e delegação aplica-se a objetos. A principal vantagem do mecanismo de delegação é facilitar o compartilhamento de comportamentos em tempo de execução.

Os conceitos de herança e delegação proporcionam uma abordagem para representar as diferentes fases de comportamento (estados) das entidades de uma determinada aplicação. Esta abordagem consiste em separar a hierarquia de classes que provê a funcionalidade específica da aplicação e criar uma hierarquia de classes para representar os diferentes estados que os objetos podem apresentar durante o seu tempo de vida. As duas hierarquias são ligadas por um mecanismo de delegação [Rub94, QR96].

À medida que a complexidade de um sistema aumenta, a construção de um modelo abstrato torna-se indispensável. A metodologia adotada para construir o modelo abstrato do nosso experimento foi a metodologia UML, que representa a unificação das metodologias de Booch [Boo91], OMT [Rum92] e OOSE [Jac92]. A UML é uma linguagem para especificação, construção, visualização e documentação dos artefatos de um sistema de computação [BRJ97]. Ela destina-se, inclusive, para modelagem de sistemas de tempo real, e sistemas distribuídos e concorrentes.

Por fim, com o objetivo de discutir questões relacionadas à reutilização de software em grande escala, apresentamos os conceitos de *framework*, padrões de projeto e metapadrões.

Um *framework* [JF88, WJ90] é um conjunto de classes abstratas e concretas, que forma uma infra-estrutura genérica de soluções para um conjunto de problemas. Ele é utilizado redefinindo-se as classes existentes ou estendendo-as através da definição de novas subclasses. Enquanto os componentes de uma biblioteca de classes são utilizados individualmente, classes em um *framework* são reutilizadas como um todo, para resolver uma instância específica de um certo problema.

Padrões de projeto [GHJV94] têm sido propostos como meio de representar, registrar e reutilizar micro-arquiteturas de projeto repetitivas, bem como a experiência acumulada pelo projetista ao desenvolver estas estruturas. Um padrão nomeia, abstrai e identifica os aspectos chaves das estruturas de projetos, identificando classes e instâncias, suas colaborações e a distribuição de responsabilidades. Eles formam uma base de experiência para construir sistemas reutilizáveis, atuando como blocos pré-fabricados para a construção de sistemas mais complexos.

Metapadrões [Pre95] descrevem como construir *frameworks* independente de um domínio específico. Eles constituem uma abordagem elegante e poderosa que pode ser aplicada para classificar e descrever padrões de projeto em um metanível.

Capítulo 3

Noções de Tolerância a Falhas e de Sistemas Distribuídos

Este capítulo apresenta uma série de definições e conceitos básicos de tolerância a falhas e de sistemas distribuídos. A Seção 3.1 descreve tolerância a falhas, incluindo alguns comentários sobre mecanismos para tolerância a falhas de hardware, falhas de software e falhas de ambiente. A Seção 3.2 descreve sistemas distribuídos, incluindo a descrição de um modelo computacional, de mecanismos para comunicação entre processos e, também, alguns comentários sobre a integração de sistemas distribuídos com o paradigma de orientação a objetos. Por fim, a Seção 3.3 apresenta um breve sumário do capítulo.

3.1 Noções de Tolerância a Falhas

Um sistema é tolerante a falhas se ele pode mascarar a presença de falhas, com base no uso de mecanismos de redundância [Jal94]. Em outras palavras, um sistema é tolerante a falhas se o seu comportamento é consistente com a sua especificação, apesar da presença de falhas em algum de seus componentes. Portanto, se algum componente do sistema torna-se falho, esta falha deve ser mascarada, de forma que não seja refletida no comportamento externo do sistema.

Existem três níveis nos quais tolerância a falhas pode ser aplicada [HW92]. Num primeiro nível, tolerância a falhas tem sido utilizada para compensar falhas decorrentes de fenômenos físicos adversos, isto é, tolerância a falhas de hardware. Através de recursos extras de hardware, um sistema aumenta sua capacidade de continuar em operação. Um

segundo nível de tolerância a falhas reconhece que uma plataforma de hardware tolerante a falhas não garante, por si só, alta disponibilidade. É importante ainda estruturar o software para compensar erros cometidos durante as fases de desenvolvimento do sistema (especificação, projeto e implementação) ou nas modificações feitas em fase de manutenção, isto é, tolerância a falhas de software. No terceiro nível, o sistema deve prover funções para compensar falhas no ambiente em que ele está inserido, ou seja, tolerância a falhas de ambiente. Por exemplo, o software pode detectar e compensar defeitos de equipamentos, erros de pessoas, falhas em sensores, etc. Medidas de tolerância a falhas neste nível são normalmente específicas da aplicação.

O nosso trabalho concentra-se principalmente no terceiro nível. Nosso objetivo é prover suporte para tolerância a falhas de ambiente, através da incorporação disciplinada de redundância, de forma que o impacto dessa redundância na complexidade do sistema possa ser mantido sob controle. Redundância é a chave para a provisão de tolerância a falhas; não pode existir tolerância a falhas sem redundância [Jal94]. Nós definimos redundância como sendo as partes de um sistema que são necessárias para o seu funcionamento somente quando consideramos a provisão de tolerância a falhas. Ou seja, o sistema funciona corretamente sem redundância quando não ocorrem falhas. Um sistema pode apresentar redundância em hardware, software ou tempo. Redundância em hardware compreende os componentes de hardware que são adicionados para que o sistema dê apoio para tolerância a falhas de hardware. Redundância em software inclui todos os programas e instruções que são empregadas para dar apoio a tolerância a falhas. Uma técnica comum para tolerância a falhas é executar alguma instrução (ou seqüência de instruções) várias vezes. Esta técnica implementa redundância de tempo, ou seja, requer tempo extra para realizar tarefas tolerantes à falhas.

As próximas Seções descrevem o impacto de falhas na complexidade de um sistema e introduzem os conceitos básicos de tolerância a falhas. A descrição dos conceitos nas próximas seções tem por base as terminologias apresentadas por Lee/Anderson [LA90] e Jalote [Jal94].

3.1.1 Conceitos Básicos

Esta seção introduz uma série de conceitos e definições importantes para o entendimento dos tipos de falhas, bem como dos mecanismos utilizados para tolerá-las.

Sistema

Um **sistema** consiste de um conjunto de componentes que interagem sobre o controle de um projeto. Componentes, por sua vez, também podem ter sub-componentes. O projeto do sistema também é um componente do próprio sistema, mas com características especiais como, por exemplo, a responsabilidade de controlar a interação entre os demais componentes do sistema. Componentes recebem requisições de serviços e produzem respostas a estas requisições (Figura 23). Com o objetivo de produzir respostas a uma dada requisição de serviço, um componente pode solicitar um determinado serviço a seus sub-componentes. Desta forma, podemos considerar um sistema como uma hierarquia de componentes, cada componente provendo algum serviço de acordo com alguma implementação [LA90].

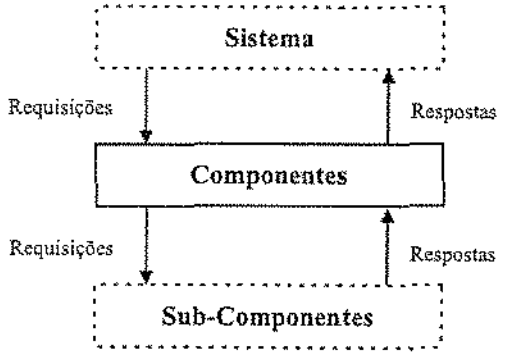


Figura 23 - Componente Geral de um Sistema

Um sistema interage com seu ambiente através de uma interface. Uma interface é simplesmente um lugar de interação entre dois sistemas. O ambiente de um sistema é um outro sistema, que provê requisições e recebe respostas do primeiro sistema, ou seja, o sistema pode prover serviços em respostas a requisições do ambiente [LA90]. Nesta dissertação, consideramos um sistema como nosso domínio de solução, o qual é cercado por um ambiente, que contém o domínio do problema. Por exemplo, se considerarmos uma

aplicação de controle de processos, de acordo com nossa terminologia, como sendo o sistema, podemos considerar como ambiente, os processos físicos ou maquinários que são mantidos e monitorados pelo controlador.

Falhas, Erros e Defeitos

Falhas (em inglês, *faults*) são imperfeições no hardware (por exemplo, elétricas, eletrônicas ou mecânicas) ou software (por exemplo, algoritmos) de um sistema de computador. Imperfeições no ambiente onde o sistema de computação está localizado também são denominadas de falhas. Um **erro** é uma parte do estado de um sistema que está incorreto ou errôneo. Um estado errôneo pode ser definido como um estado que pode conduzir a defeitos. Um **defeito** (em inglês, *failure*) ocorre quando o comportamento do sistema desvia-se do comportamento esperado (definido na sua especificação).

Temos, portanto, a relação falha -> erro -> defeito. Quando há uma falha no sistema, sua manifestação pode dar origem a erros, os quais podem resultar em um subseqüente defeito no sistema.

Prevenção de Falhas e Tolerância a Falhas

Existem duas abordagens distintas para a provisão de sistemas confiáveis: prevenção de falhas e tolerância a falhas.

Técnicas de **prevenção de falhas** tentam assegurar que o sistema não contém ou não conterá falhas. Para tanto, técnicas são utilizadas para evitar a introdução de falhas durante o projeto e construção do sistema. Além disso, técnicas de remoção de falhas como teste e validação, são usadas para melhorar a confiabilidade total do sistema, através da remoção de falhas presentes no mesmo. Infelizmente, é difícil (para sistemas complexos, praticamente impossível) remover todas as falhas. Desta forma, técnicas de tolerância a falhas são requeridas para tratar as falhas residuais na fase operacional de um sistema.

Técnicas de **tolerância a falhas** tentam evitar que falhas e erros causem defeitos no sistema. Quando falhas se manifestam, estados errôneos aparecem no sistema de forma que a primeira fase de tolerância a falhas é a **detecção de erros**. Uma vez que um erro foi detectado, implica que uma falha em algum componente ocorreu. Qualquer dano causado

pela falha entre a manifestação de um erro e sua detecção é identificado e delimitado na fase de **confinamento e avaliação**. As duas últimas fases são a recuperação de erros e o tratamento de falhas. Técnicas de **recuperação de erros** têm como objetivo transformar o estado corrente e incorreto do sistema para um estado bem definido e livre de erros, a partir do qual a operação normal do sistema pode continuar. Técnicas de **tratamento de falhas** identificam os componentes falhos e asseguram que a falha manifestada não se repita imediatamente.

Tratamento de Exceções

Exceções são definidas como erros ou situações excepcionais que surgem durante a execução de um programa. Mecanismos de tratamento de exceções permitem a definição de condições excepcionais (isto é, exceções) e de tratadores para tais exceções. Quando uma exceção é detectada, o processamento normal é suspenso e o controle é transferido para o tratador associado a ela.

Em tolerância a falhas, exceções e mecanismos de tratamento de exceções são usados para conectar ações de componentes do sistema que pertencem a diferentes níveis de abstração. A atividade de um componente do sistema pode ser dividida em duas partes: uma parte normal e uma parte anormal (de tratamento de exceções). A parte normal implementa o serviço normal do componente e a parte de tratamento de exceções implementa os recursos para tolerar as falhas que causam tais exceções.

Exceções podem ser classificadas em três grupos: exceções de interface, exceções internas e exceções de falha. Uma **exceção de interface** é sinalizada em resposta a uma requisição de um serviço não disponível ou um serviço invocado com um conjunto inválido de parâmetros. Uma possível causa desse grupo de exceções é uma falha no projeto do componente. Uma **exceção interna** é gerada quando um componente detecta uma situação inesperada durante sua atividade normal. Se esta exceção não puder ser tratada pelo próprio componente, então uma **exceção de falha** é levantada, indicando que, por algum motivo, este componente não pode prover o serviço especificado. Quando um componente recebe uma resposta anormal de um outro componente (exceção de interface ou exceção de falha) ou detecta uma condição anormal durante sua execução (exceção interna), ele invoca os mecanismos apropriados de tolerância a falhas. Se estas exceções são tratadas então o

componente pode voltar a prover o serviço normal. Entretanto, se o componente não consegue tratar a exceção, uma exceção de falha é sinalizada.

Um componente tolerante a falhas ideal é aquele que possui uma interface bem definida, incluindo respostas normais e anormais nesta interface de interação, como mostra a Figura 24.

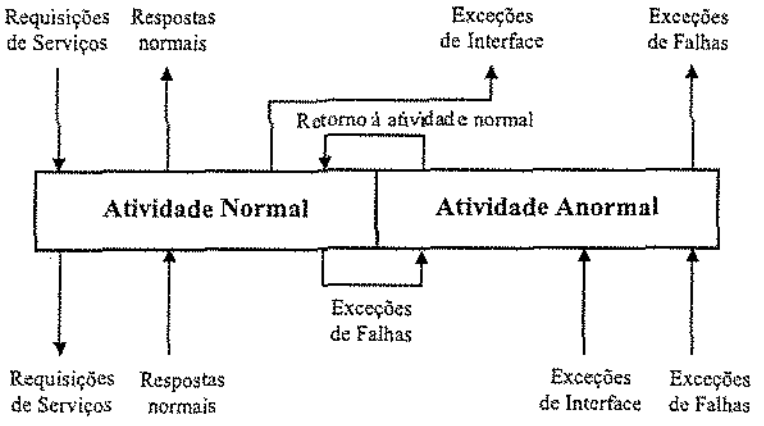


Figura 24 - Componente Tolerante a Falhas Ideal

Mecanismos de Recuperação de Erros

Como vimos, uma vez que o erro foi detectado (fase de detecção de erros) e sua extensão identificada (fase de confinamento e avaliação), o mesmo precisa ser removido (fase de recuperação de erros). Na fase de recuperação de erros, o sistema é posto em um estado livre de erros, isto é, o sistema é posto em um estado consistente. Existem dois mecanismos básicos para a recuperação de erros: recuperação de erros por avanço e recuperação de erros por retrocesso. **Recuperação por avanço** manipula algumas partes do estado corrente do sistema para produzir um novo estado livre de erros. **Recuperação por retrocesso** restaura o sistema para um estado anterior, descartando o estado corrente e incorreto.

Exceções e tratamento de exceções constituem um exemplo de mecanismo aplicado para a provisão de recuperação de erros por avanço. Por outro lado, o esquema de blocos de recuperação (descrito na seção 3.1.3) provê uma estrutura que dá apoio à recuperação de erros por retrocesso. Deste modo, tratamento de exceções e blocos de recuperação são abordagens complementares para implementar recuperação de erros em um sistema.

Ações Atômicas

O padrão de interação entre os componentes de um sistema pode ser estruturado através do conceito de ações atômicas. A atividade de um grupo de componentes constitui uma ação atômica se não existe interação entre o grupo e o resto do sistema durante a atividade [LA90]. Para o resto do sistema, todas as atividades dentro de uma ação atômica apresentam-se como indivisíveis.

Em sistemas tolerantes a falhas baseados neste modelo, as operações são executadas como ações atômicas com as seguintes propriedades:

(i) Equivalência Serial: esta propriedade garante que se duas ou mais ações estão executando concorrentemente sobre objetos compartilhados, o resultado final é equivalente a uma execução das ações em alguma ordem serial;

(ii) Atomicidade: esta propriedade garante que uma ação ou é consolidada, terminando normalmente e gerando o resultado desejado, ou então é abortada e não produz resultado;

(iii) Permanência de Efeito: esta propriedade garante que o resultado gerado por uma ação consolidada, torna-se permanente e nenhuma falha que possa vir a acontecer posteriormente pode desfazê-lo ou causar a sua perda;

3.1.2 Falhas de Hardware

Ao contrário de software, que não tem nenhuma propriedade física e, portanto, nenhuma causa física de falhas, falhas de hardware são originadas por causas físicas (por exemplo, panes em computadores e problemas de comunicação na rede). Falhas de hardware podem ser introduzidas durante o processo de fabricação dos componentes de hardware ou podem ocorrer durante a fase operacional do sistema [Jal94]. Porém, estes dois tipos não são independentes; problemas no processo de fabricação podem afetar a fase operacional. Técnicas de tolerância a falhas de hardware tratam da ocorrência e do efeito destes problemas físicos nos componentes de hardware.

O passo inicial para a obtenção de tolerância a falhas de hardware é utilizar redundância de hardware, empregada na forma de recursos extras como, por exemplo,

processadores, memória e recursos de comunicação em rede. Além disso, existem várias abordagens para dar suporte a tolerância a falhas de hardware. Uma abordagem consiste em usar mecanismos de linguagens orientadas a objetos. Por exemplo, o sistema Arjuna [Shr95], que é um sistema de programação orientada a objetos, provê um conjunto de ferramentas com o objetivo de auxiliar a construção de aplicações distribuídas e tolerantes a falhas. Todo o sistema Arjuna é baseado no modelo de objetos e ações. Além disso, ele implementa transações atômicas encadeadas, controle de concorrência, recuperação a nível de objeto e persistência. Todos estes conceitos formam a base para tornar uma aplicação tolerante a falhas de hardware.

3.1.3 Falhas de Software

Como vimos, a causa básica de falhas nos componentes de hardware é normalmente alguma falha física. Software, em contrapartida, não tem propriedades físicas; é uma entidade totalmente conceitual. Portanto, falhas físicas não existem em software. Falhas de software são sempre falhas cometidas durante a fase de desenvolvimento (especificação, projeto e implementação) ou nas modificações realizadas em fase de manutenção do sistema. Tolerância a falhas de software compreende as técnicas necessárias para tolerar estas falhas decorrentes da construção e manutenção do próprio software. Dois principais métodos têm sido propostos para tolerar falhas de software: N-versões e Blocos de Recuperação [LA90].

N-versões consiste de vários programas desenvolvidos independentemente e que satisfazem uma especificação comum. As versões executam simultaneamente e seus resultados são comparados por algum tipo de avaliação. Baseado no voto da maioria, esta avaliação pode eliminar resultados incorretos e passar o resultado gerado pela maioria para o resto do sistema.

O esquema de **Blocos de Recuperação** possui a seguinte sintaxe:

```
ensure TESTE DE ACEITAÇÃO
by MÓDULO 1
.
.
.
else by MÓDULO N
else error
```

No início da execução de um bloco de recuperação, o estado do sistema é guardado através de pontos de recuperação. O módulo 1 é executado e o teste de aceitação avalia as saídas produzidas pelo módulo. Se este passar pelo teste, o controle volta para a próxima instrução após o bloco. Caso contrário, gera-se uma exceção e o sistema volta para o estado em que se encontrava antes de entrar no bloco. Em seguida o próximo módulo é executado e o teste de aceitação é aplicado novamente. Se todos os módulos disponíveis falharem, o bloco gerará um erro. O sucesso do bloco de recuperação depende do teste de aceitação, que deve ser simples o suficiente para não introduzir sobrecargas no tempo de execução.

3.1.4 Falhas de Ambiente

Muitas entidades no domínio de problemas do mundo real podem mudar seus comportamentos devido a uma grande variedade de fenômenos como, por exemplo, defeitos de equipamentos do ambiente, erros de pessoas, falhas em sensores, etc. **Falhas de Ambiente** referem-se às mudanças infreqüentes, indesejáveis e imprevisíveis no comportamento das entidades do ambiente. Além disso, mesmo quando todas as possíveis mudanças das entidades relacionadas ao domínio do problema são previstas a princípio, pode ser desejável representar tais mudanças explicitamente no domínio da solução, em vez de simplesmente codificá-las a nível de implementação.

Técnicas de programação orientadas a objetos são bastante apropriadas para prover tolerância a falhas de ambiente. A nossa abordagem considera um objeto no domínio da solução como "falho" se seu comportamento é alterado devido a falhas de ambientes ocorridas no domínio do problema. Com isso, o comportamento do objeto muda de normal para anormal e, portanto, o objeto deve prover o tratamento de falhas adequado, mudando a implementação de seus métodos. Para tanto, propomos a utilização de conceitos como delegação, classes abstratas e hierarquia de estados [Rub94, QR96]. No próximo capítulo, introduzimos um estudo de caso detalhado, onde estes conceitos são aplicados para tolerar falhas de ambiente.

3.2 Noções de Sistemas Distribuídos

Sistemas Distribuídos consistem fisicamente de uma arquitetura caracterizada pela interligação de máquinas autônomas com capacidades próprias de armazenamento e processamento, compartilhando recursos de maneira transparente, eficiente e segura. Logicamente, estes sistemas caracterizam-se pela descentralização de controle entre os vários componentes de software que executam independente e cooperativamente, trocando informações e oferecendo serviços uns aos outros.

O paradigma de orientação a objetos é bastante apropriado para o desenvolvimento de sistemas distribuídos. Em nosso trabalho, utilizamos conceitos como o de objetos e de composição de objetos, para representar as unidades de distribuição e encapsular os mecanismos de comunicação entre estas unidades. Estas idéias são ilustradas no estudo de caso descrito no próximo capítulo.

Na próxima Seção descrevemos o modelo computacional que melhor se adapta ao nosso ambiente distribuído, incluindo uma classificação para os tipos de falhas que podem ocorrer neste ambiente. Em seguida, comentamos o padrão de comunicação a ser utilizado em nosso experimento e fazemos alguns comentários sobre a integração dos conceitos de orientação a objetos e sistemas distribuídos.

3.2.1 Modelo Computacional

Existem dois modos principais de visualizarmos um sistema distribuído: (1) através de seu modelo físico e (2) através de seu modelo lógico. O modelo físico corresponde aos componentes físicos do sistema e o modelo lógico corresponde ao processamento ou computação. O modelo lógico é importante, pois corresponde aos serviços que são definidos para atender as perspectivas do usuário. O modelo físico é importante também, uma vez que a computação é realizada sobre uma configuração física [Jal94].

Modelo Físico

A configuração física de um sistema distribuído consiste de vários computadores (ou nodos) que estão em diferentes localizações, mas são conectados por uma rede de

comunicação. Todos os nodos são autônomos e comunicam-se entre si através da rede de comunicação. Portanto, duas propriedades fundamentais de sistemas distribuídos são a separação geográfica e a natureza autônoma dos vários nodos.

Sistemas distribuídos são diferentes de sistemas paralelos. Em sistemas paralelos, os nodos são fortemente acoplados, isto é, não são autônomos. Ao contrário, os elementos computacionais de um sistema distribuído são desacoplados. Uma outra característica importante de sistemas distribuídos é a ausência de memória compartilhada entre diferentes nodos. Uma posição de memória pertencente a um determinado nodo só pode ser acessada pelo próprio nodo. Em contrapartida, alguns sistemas paralelos têm memória compartilhada.

Uma outra diferença entre sistemas paralelos e sistemas distribuídos é que sistemas distribuídos não têm um relógio global conduzindo todos os nodos, ao passo que sistemas paralelos geralmente têm um único relógio no sistema. Em sistemas distribuídos, cada nodo tem seu próprio relógio, que pode ser utilizado para controlar as instruções executadas neste nodo. Entretanto, o relógio de um nodo não pode controlar diretamente as instruções de um outro nodo. Um sistema paralelo pode ter um relógio global que é utilizado para controlar instruções de diferentes elementos do sistema.

Em síntese, o nosso trabalho supõe como modelo físico, um sistema distribuído consistindo de vários nodos. Cada nodo consiste de um processador, que tem alguma memória privada que é inacessível aos outros nodos, e um relógio privado que conduz a execução de instruções deste processador. Cada nodo tem uma interface de rede, através da qual ele é conectado à rede de comunicação. Dois nodos do sistema comunicam-se entre si através de troca de mensagens sobre a rede de comunicação. Além disso, existe o software que controla a seqüência de instruções a serem executadas em cada nodo. Portanto, os principais componentes de um sistema distribuído são os processadores, a rede de comunicação, os relógios e os softwares.

Modelo Lógico

Uma aplicação distribuída (ou um sistema distribuído sob o ponto de vista da aplicação) consiste de um conjunto de processos executando e cooperando uns com os outros para realizar uma determinada tarefa. Um processo representa a execução de um

programa seqüencial; uma lista de declarações e/ou instruções. Processos concorrentes podem ser executados num mesmo processador, permitindo o compartilhamento dos recursos deste processador por processos diferentes. Esta abordagem é conhecida por multiprogramação. Em sistemas distribuídos, entretanto, supomos o caso onde diferentes processos executam em paralelo em diferentes nodos de um sistema. Desta forma, no nível lógico, consideramos um sistema distribuído consistindo de um conjunto finito de processos e canais entre os processos. Canais representam a conexão lógica entre os processos. Um canal entre dois processos existe se estes interagem através de troca de mensagens.

Classificação de Falhas

Uma maneira de classificarmos as falhas que podem ocorrer num sistema distribuído consiste na avaliação de como um componente se comporta quando ele falha. Esta classificação especifica quais as suposições podem ser feitas a respeito do comportamento do componente quando ocorre uma falha. De acordo com esta abordagem, podemos classificar as falhas em quatro principais categorias [Cri91, Jal94]:

- **falha por queda**, que causa parada ou perda do estado interno em um componente;
- **falha por omissão**, que causa em um componente a omissão de respostas para determinadas entradas;
- **falha por tempo**, que ocorre quando a resposta do componente é funcionalmente correta mas não obedece o intervalo de tempo real especificado: falha por antecedência ou falha por atraso;
- **falha arbitrária** (*byzantine*), quando qualquer comportamento falho é permitido para um determinado componente, isto é, o componente comporta-se de maneira totalmente arbitrária quando uma falha ocorre;

Estas falhas formam uma hierarquia, sendo que a falha por queda é a mais simples e, portanto, mais restritiva e a falha arbitrária é a menos restritiva. Elas têm um relacionamento de inclusão, como mostra a Figura 25.

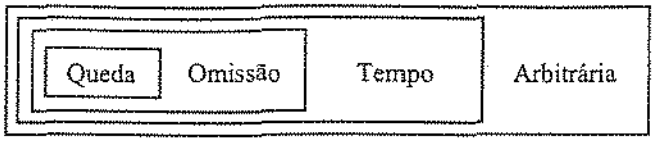


Figura 25 - Classificação de Falhas

Existe uma outra categoria de falhas que pode ser adicionada a este grupo, denominada **falha por computação incorreta**. Esta falha também é um subconjunto de falhas arbitrárias, mas é diferente de todas as outras categorias. Com este tipo de falha, um componente produz uma saída incorreta em resposta a determinadas entradas.

Para um processador, supomos mais freqüentemente: uma falha por queda, quando ele pára a execução ou uma falha arbitrária, quando nenhuma suposição é feita sobre seu comportamento falho. Em uma rede de comunicação, por outro lado, os vários tipos de falhas são normalmente considerados. A rede de comunicação pode sofrer uma falha por queda, quando não faz nada, ou seja, não recebe nem distribui mensagens; pode corromper mensagens (falha por computação incorreta); pode perder mensagens (falha por omissão); pode distribuir mensagens atrasadas ou antecipadas (falha por tempo); e pode comportar-se de uma forma totalmente arbitrária (falha arbitrária). No caso do relógio, a falha que tipicamente nos interessa é quando o relógio "corre" muito rápido ou muito devagar (falha por tempo). Além disso, podemos considerar também uma falha arbitrária, quando o comportamento do relógio (isto é, o tempo que ele representa) é arbitrário. Um relógio que pára e mostra sempre o mesmo tempo pode ser considerado como sofrendo uma falha por omissão. Por fim, os componentes de software podem ter a maioria das falhas descritas. Entretanto, a falha de software que mais interessa é a falha por computação incorreta. Na presença desta falha, o software realiza operações erradas. Neste contexto, esta falha é conhecida também como falha de projeto ou falha de software.

3.2.2 Comunicação entre Processos

Como vimos, os componentes de um sistema distribuído são lógica e fisicamente separados; eles precisam comunicar-se entre si para interagir e realizar tarefas. Nós supomos que todos os componentes que requerem e provêem acesso para recursos em um sistema distribuído são implementados como processos. Portanto, os processos se

comunicam entre si para realizar tarefas. A comunicação entre um par de processos é realizada através de primitivas para troca de mensagens — envia e recebe — que juntas resultam na:

a) transferência de dados do ambiente do processo que envia a mensagem (remetente) para o ambiente do processo que recebe a mensagem (destinatário);
b) sincronização entre o processo remetente e o processo destinatário;

Cada troca de mensagem envolve a transmissão de um conjunto de dados (uma mensagem) por intermédio de um determinado mecanismo de comunicação. Este mecanismo pode ser síncrono, o que implica que o processo remetente espera após a transmissão de uma mensagem até que o processo destinatário tenha aceito a mensagem ou pode ser assíncrono, o que implica que a mensagem é colocada numa fila de mensagens esperando pela aceitação do processo destinatário e o processo remetente pode prosseguir imediatamente [CDK94].

Sistemas distribuídos podem ser projetados inteiramente em termos de troca de mensagens, mas existem certos padrões de comunicação que podem ser considerados como essenciais para o suporte ao projeto e à construção de um sistema distribuído. Em particular, temos o modelo de comunicação cliente-servidor, que é um dos modelos de comunicação mais utilizados para a construção de sistemas distribuídos. Na próxima seção introduzimos algumas idéias básicas do modelo cliente-servidor. Em seguida, descrevemos o mecanismo de chamada de procedimento remoto, que é um mecanismo bastante utilizado para implementar o modelo cliente-servidor a nível de linguagem de programação.

3.2.2.1 Comunicação Cliente-Servidor

O modelo de comunicação cliente-servidor é orientado para provisão de serviço [CDK94]. Uma interação entre um processo cliente e um processo servidor consiste:

(i) da transmissão de uma requisição de serviço do cliente para o servidor;

(ii) da execução do serviço pelo servidor e

(iii) da transmissão da resposta para o cliente.

Esse padrão de comunicação envolve a transmissão de duas mensagens (Figura 26) e uma forma específica de sincronização entre cliente e servidor. O cliente invoca um serviço

enviando uma mensagem de requisição para o servidor (passo 1). O servidor realiza o serviço requisitado (passo 2) e envia uma mensagem de resposta para o cliente (passo 3). Após o envio da requisição (passo 1), a atividade do cliente fica bloqueada até que a resposta seja recebida.

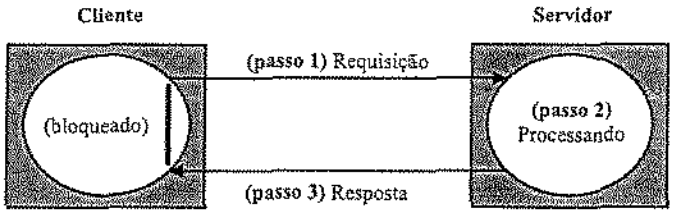


Figura 26 - Comunicação Cliente-Servidor

Normalmente a comunicação é síncrona, pois o cliente bloqueia até que uma resposta do servidor seja recebida. Entretanto, comunicação assíncrona é uma alternativa que pode ser útil em situações onde clientes continuam a execução e deixam para receber a resposta posteriormente.

Uma observação importante é que um processo é cliente ou servidor somente para o propósito de uma troca particular ou uma série de trocas. Entretanto, um servidor pode requisitar serviços de outro servidor, podendo, portanto, ser cliente de outros processos, e similarmente, um cliente pode ser um servidor para outros processos.

3.2.2.2 Chamada de Procedimento Remoto

Chamada de Procedimento Remoto (em inglês, *RPC - Remote Procedure Call*) [BN84] integra o modelo cliente-servidor com o mecanismo de chamada de procedimento local, permitindo que os clientes se comuniquem com os servidores através da chamada de procedimento de uma maneira semelhante ao uso convencional de chamada de procedimentos das linguagens de programação de alto nível. Uma chamada de procedimento remoto é modelada como uma chamada de procedimento local, mas o procedimento chamado é executado por um processo diferente, normalmente em outro espaço de endereçamento.

O mecanismo de chamada de procedimento remoto é elegante e oferece simplicidade para o programador, pois encapsula detalhes como estabelecimento da comunicação entre

cliente e servidor, empacotamento de parâmetros nas mensagens e desempacotamento de resultados. Desta forma, RPC mantém o máximo possível a semântica de chamada de procedimento local, porém num ambiente de implementação bastante diferente.

A Figura 27 ilustra os principais passos realizados na chamada de um procedimento remoto [Tan92]. Eles podem ser resumidos da seguinte maneira:

1. O procedimento cliente chama o *stub* cliente. O propósito de um procedimento *stub* é converter a chamada de procedimento local para uma chamada de procedimento remoto. Os tipos de parâmetros empacotados pelo *stub* cliente devem estar em conformidade com os esperados pelo procedimento remoto. Este requisito é cumprido através da utilização de uma linguagem comum de definição de interface;
2. O *stub* cliente constrói a mensagem e a envia sobre a rede de comunicação. Para tanto, utiliza o núcleo do sistema operacional (*kernel*);
3. O *stub* servidor desempacota os parâmetros e chama o servidor;
4. O servidor realiza o serviço e retorna a resposta para o *stub* servidor;
5. O *stub* servidor empacota a resposta em uma mensagem e envia-a sobre a rede de comunicação. Novamente utilizando o *kernel*;
6. O *stub* cliente desempacota a resposta e retorna para o cliente;

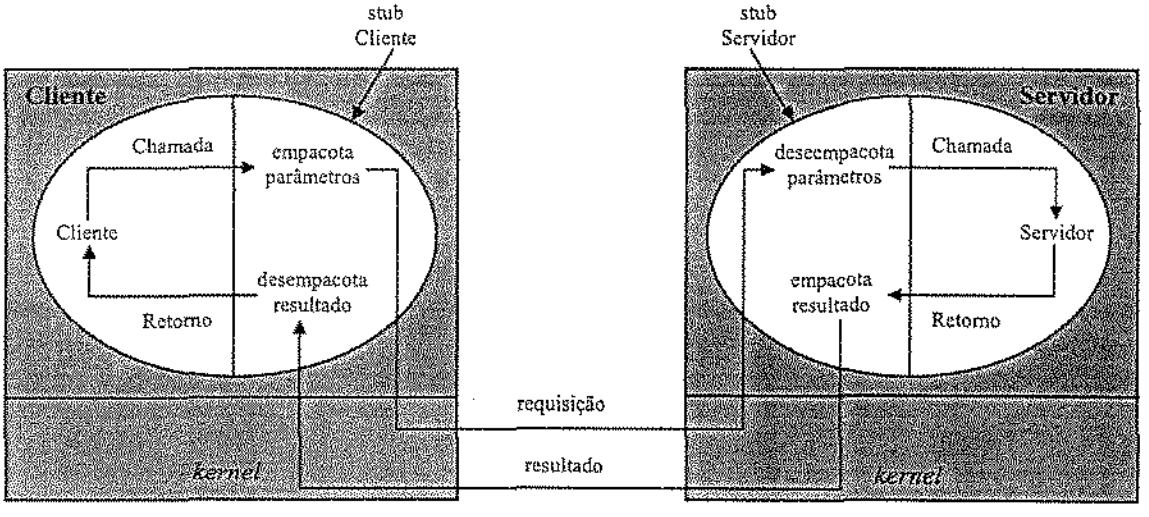


Figura 27 - Chamada de Procedimento Remoto

3.2.3 Sistemas Distribuídos e Orientação a Objetos

O paradigma de orientação a objetos (descrito no Capítulo 2) é bastante apropriado para o desenvolvimento de sistemas distribuídos. Ele provê um suporte natural para as características necessárias para a construção de sistemas distribuídos bem estruturados. Além disso, objetos ou grupos de objetos (composições) provêem uma boa base para o particionamento do sistema em componentes que podem ser distribuídos sobre uma rede [LRH96].

Uma motivação para a união dos conceitos de sistemas distribuídos com o paradigma de objetos decorre da similaridade entre as suas construções básicas, destacando-se a analogia que pode ser feita entre objetos e processos [Mey93]. Dentre outras características, os dois modelos dão suporte para:

- variáveis locais (atributos de uma classe; variáveis de um processo);
- comportamento encapsulado (métodos de uma classe; ciclos de um processo);
- mecanismo de comunicação baseado em troca de mensagens;

Em sistemas orientados a objetos, um objeto tem uma interface pública composta por um conjunto de operações (ou métodos). Esses métodos são visíveis para os outros objetos ou processos externos a este objeto. Este modelo se adapta naturalmente a sistemas distribuídos. Um objeto pode residir em diferentes nodos. Um processo, que deseja executar um método num objeto particular, envia uma mensagem para o objeto, o qual executa o método (se este é acessível externamente) e retorna o resultado. Isto equivale, em sistemas distribuídos, a uma chamada de procedimento remoto, exceto que o procedimento é um método particular de um determinado objeto e trabalha somente naquele objeto [Jal94].

Na literatura encontramos vários exemplos de sistemas de suporte para programação distribuída orientada a objetos, tal como, o sistema Arjuna [Shr95], que é um sistema de programação orientada a objetos, totalmente implementado em C++, que provê um conjunto de ferramentas que auxiliam a construção de aplicações distribuídas e tolerantes a falhas, estruturadas como ações atômicas operando sobre objetos persistentes. Toda a funcionalidade do sistema Arjuna é fornecida através de classes, ou seja, além de suportar o

modelo de computação orientada a objetos, toda a sua estrutura interna é também orientada a objetos. Isto permite que os objetos definidos pelo usuário, por meio de mecanismos de herança, incorporem essa funcionalidade (como por exemplo, persistência e atomicidade).

Metodologia de Análise e Projeto

Para aplicar-se os princípios de orientação a objetos a um ambiente distribuído, as metodologias de análise e projeto orientadas a objetos devem integrar as noções de distribuição e fornecer notações que dão suporte a esses novos requisitos [LRH96]. A UML (Seção 2.3.1) provê uma série de elementos destinados à modelagem de sistemas distribuídos como, por exemplo:

- diagramas de casos de usos podem ser utilizados na especificação de requisitos, incluindo informações de localização dos componentes, concorrência no domínio do problema, confiabilidade e extensibilidade;
- diagramas de colaborações, incluindo os conceitos de objetos ativos e estereótipos, podem ser utilizados para representar o modelo lógico de um sistema distribuído;
- diagramas de componentes e diagramas de disposições, incluindo o conceito de estereótipo, podem ser utilizados para representar o modelo físico de um sistema distribuído;

3.3 Sumário

Noções de Tolerância a Falhas

Um sistema consiste de um conjunto de componentes que interagem sobre o controle de um projeto. Um sistema é tolerante a falhas quando o seu comportamento é consistente com a sua especificação, apesar da presença de falhas em algum de seus componentes. Falhas podem ser classificadas em três níveis: falhas de hardware, que são originadas por causas físicas; falhas de software, que são cometidas durante a fase de desenvolvimento ou manutenção do software; e, falhas de ambiente, que ocorrem no ambiente em que o sistema está inserido como, por exemplo, falhas em sensores. Esta dissertação concentra-se em

tolerância a falhas de ambiente, através do uso de conceitos de orientação a objetos como delegação, classes abstratas e hierarquia de estados.

Existe uma importante distinção entre os conceitos de falha, erro e defeito. Quando existe uma falha no sistema, a manifestação desta falha pode causar erros no estado do sistema, podendo resultar num subseqüente defeito no sistema. O objetivo dos mecanismos de tolerância a falhas é prevenir que erros e falhas levem o sistema a um estado defeituoso. Para tanto, quatro fases podem ser identificadas: (1) detecção de erros, (2) confinamento e avaliação, (3) recuperação de erros e (4) tratamento de falhas.

Exceções e mecanismos de tratamento de exceções são necessários para conectar ações de componentes do sistema que pertencem a diferentes níveis de abstração do sistema. Exceções são definidas como erros ou situações excepcionais que surgem durante a execução de um programa. Mecanismos de tratamento de exceções permitem a definição de condições excepcionais (isto é, exceções) e tratadores para essas exceções. Quando uma exceção é detectada, o processamento normal é suspenso e o controle é transferido para o tratador associado a ela.

O padrão de interação entre os componentes de um sistema pode ser estruturado através do conceito de ações atômicas. A atividade de um grupo de componentes constitui uma ação atômica se não existe interação entre o grupo e o resto do sistema durante a atividade [LA90].

Noções de Sistemas Distribuídos

O modelo computacional de um sistema distribuído consiste de duas partes: (1) modelo físico e (2) modelo lógico. O nosso trabalho supõe como modelo físico, um sistema distribuído consistindo de vários nodos. Cada nodo consiste de um processador, que tem alguma memória privada que é inacessível pelos outros nodos, e um relógio privado que conduz a execução de instruções deste processador. Cada nodo tem uma interface de rede, através da qual ele é conectado à rede de comunicação. Dois nodos do sistema comunicam-se entre si através de troca de mensagens sobre a rede de comunicação. Do ponto de vista lógico, um sistema distribuído consiste de um conjunto de processos executando em diferentes nodos e cooperando uns com os outros para realizar uma determinada tarefa.

Falhas em sistemas distribuídos podem ser classificadas em: falha por queda, que causa parada ou perda do estado interno em um componente; falha por omissão, que causa em um componente a omissão de respostas para determinadas entradas; falha por tempo, que ocorre quando a resposta do componente não obedece o intervalo de tempo real especificado; e, falha arbitrária, quando qualquer comportamento de falha é permitido para um determinado componente.

Sistemas distribuídos podem ser projetados inteiramente em termos de troca de mensagens, mas existem certos padrões de comunicação essenciais para o suporte ao projeto e à construção de um sistema distribuído. Em nosso experimento, utilizamos o padrão de comunicação cliente-servidor, que é implementado a nível de linguagem, utilizando o mecanismo de chamada de procedimento remoto (RPC). Chamada de procedimento remoto consiste numa abstração para comunicação cliente-servidor, com o objetivo de esconder as camadas da rede, de forma que pareça que estamos usando uma interface convencional de chamada de procedimento local. RPC é elegante e oferece simplicidade para o programador, pois encapsula detalhes como estabelecimento da comunicação entre cliente e servidor, empacotamento de parâmetros nas mensagens e desempacotamento de resultados.

O paradigma de orientação a objetos é bastante apropriado para o desenvolvimento de sistemas distribuídos. Em nosso trabalho, utilizamos conceitos como o de objetos e o de composição de objetos, para representar as unidades de distribuição e encapsular os mecanismos de comunicação entre estas unidades. Estas idéias são ilustradas no estudo de caso descrito no próximo capítulo.

Capítulo 4

Estudo de Caso: Um Controlador de Trens

A melhor maneira de entender as técnicas descritas anteriormente é utilizá-las numa aplicação prática. Portanto, este capítulo descreve a utilização de técnicas de orientação a objetos para a estruturação de uma aplicação distribuída e confiável — um Controlador de Trens.

A abordagem utilizada consiste na construção de um sistema para controlar um ambiente complexo e sujeito a erros: um modelo de ferrovia composta por vários conectores (agulhas de desvio) e sensores espalhados nos trilhos. Este modelo de ferrovia não corresponde ao domínio completo de uma malha ferroviária moderna; podemos interpretá-lo como um subdomínio — um modelo simplificado de sistema de controle e monitoração. Nesta dissertação nos referimos a este subdomínio como Sistema Controlador de Trens, ou simplesmente, Controlador de Trens.

Além da funcionalidade básica do subdomínio, dois outros requisitos fundamentais são atendidos pelo Controlador de Trens:

1) Prover controles distribuídos para as diversas partes da malha ferroviária. Neste caso, quando o controlador de uma parte A da malha precisa se comunicar com o controlador de uma parte B, utilizamos um objeto intermediário (conhecido como *proxy*), que implementa o protocolo de comunicação entre os dois controladores distribuídos e, com isso, permite o acesso transparente a um objeto em outro espaço de endereçamento.
2) Assegurar que a operação correta e confiável do sistema seja mantida (não há

colisão de trens) apesar da presença de falhas de ambiente, tais como defeitos em conectores e sensores. Neste caso, a idéia chave consiste em identificar os objetos da aplicação que alteram o seu comportamento em tempo de execução, devido a mudanças no ambiente externo. Para estes objetos, uma hierarquia de classes que captura os diferentes estados é construída em paralelo com a hierarquia de classes da aplicação propriamente dita. As duas hierarquias são ligadas por um mecanismo de delegação.

Um trabalho extenso sobre a aplicação de técnicas de programação orientada a objetos para tolerância a falhas de ambiente pode ser encontrado em [Rub94]. O nosso experimento assume esse trabalho como ponto de partida e estende a abordagem para uma plataforma distribuída. Esta plataforma é composta fisicamente por um conjunto de estações UNIX, conectadas através de uma rede TCP/IP. Logicamente, o sistema é organizado através do modelo cliente-servidor.

Este capítulo é organizado da seguinte maneira: Seção 4.1 apresenta a especificação do sistema; Seção 4.2 descreve os requisitos funcionais do sistema; Seção 4.3 descreve o modelo de análise; em seguida, Seção 4.4 aborda os aspectos de tolerância a falhas de ambiente; Seção 4.5 apresenta alguns aspectos de implementação dos requisitos de distribuição e tolerância a falhas; e, por fim, a Seção 4.6 apresenta algumas conclusões.

4.1 Especificação do Sistema

O Controlador de Trens (Figura 28) é um sistema de controle e monitoração de um modelo de ferrovia. Este modelo é composto por uma malha ferroviária e por um conjunto de trens. A malha ferroviária (Figura 29) é montada em três partes separadas, que são controladas por computadores independentes, ligados em rede. Cada parte da malha é composta por conectores, sensores e trilhos (que ligam conectores e sensores). Os trens, sensores e conectores são conectados a decodificadores, que constituem a interface para troca de sinais com o sistema.

Sensores são dispositivos que detectam a presença de trens e representam a única fonte de informação sobre o estado da ferrovia. Porém, sensores não são dispositivos confiáveis,

pois às vezes podem ser ativados erroneamente. Além disso, conectores estão sujeitos a falhas eletro-mecânicas e, conseqüentemente, trens podem desviar do trajeto predefinido.

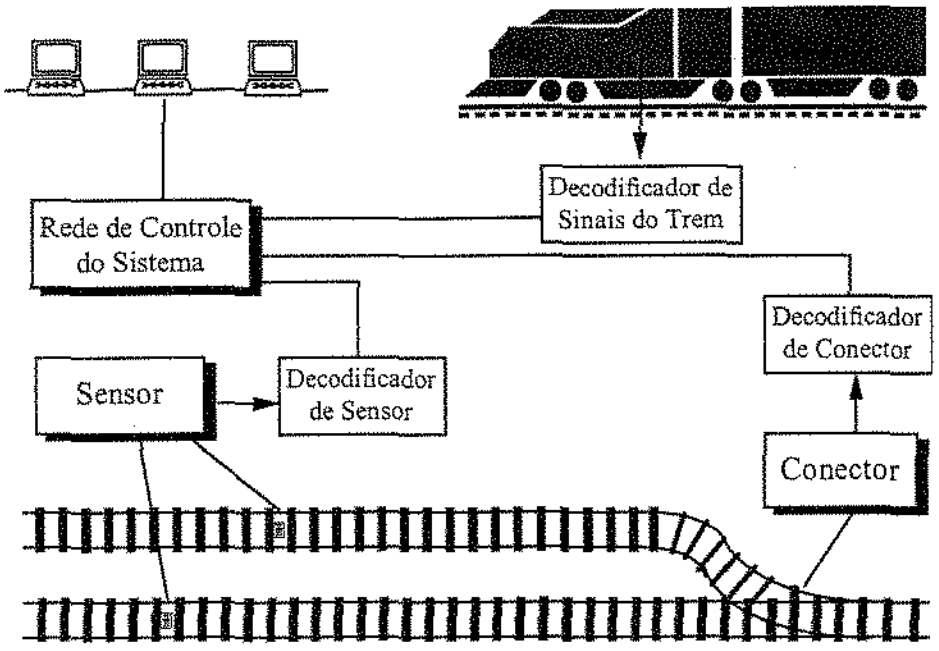


Figura 28 - Componentes do Controlador de Trens

Cada ponto na malha onde se localiza um sensor representa uma estação ferroviária. A ligação direcionada entre duas estações adjacentes forma uma seção da malha. Em outras palavras, uma seção é delimitada por duas estações adjacentes e pode conter um ou mais conectores.

Os trens se movem aleatoriamente entre estações. Isto significa que um trem inicia o trajeto em uma determinada seção da malha e continua o percurso de acordo com a disponibilidade das próximas seções. Se a próxima seção do trajeto estiver reservada para a passagem de outro trem, uma outra seção disponível é localizada e assim por diante. Apesar da presença de falhas em conectores e sensores, os trens devem mover-se pela ferrovia sem colisões podendo, se necessário, parar e reverter a direção.

O Controlador de Trens é uma aplicação distribuída; as três partes da malha são controladas por computadores independentes, ligados em rede. Uma seção pode, portanto, localizar-se na fronteira entre duas partes da malha e os trens, por sua vez, podem atravessar essa fronteira.

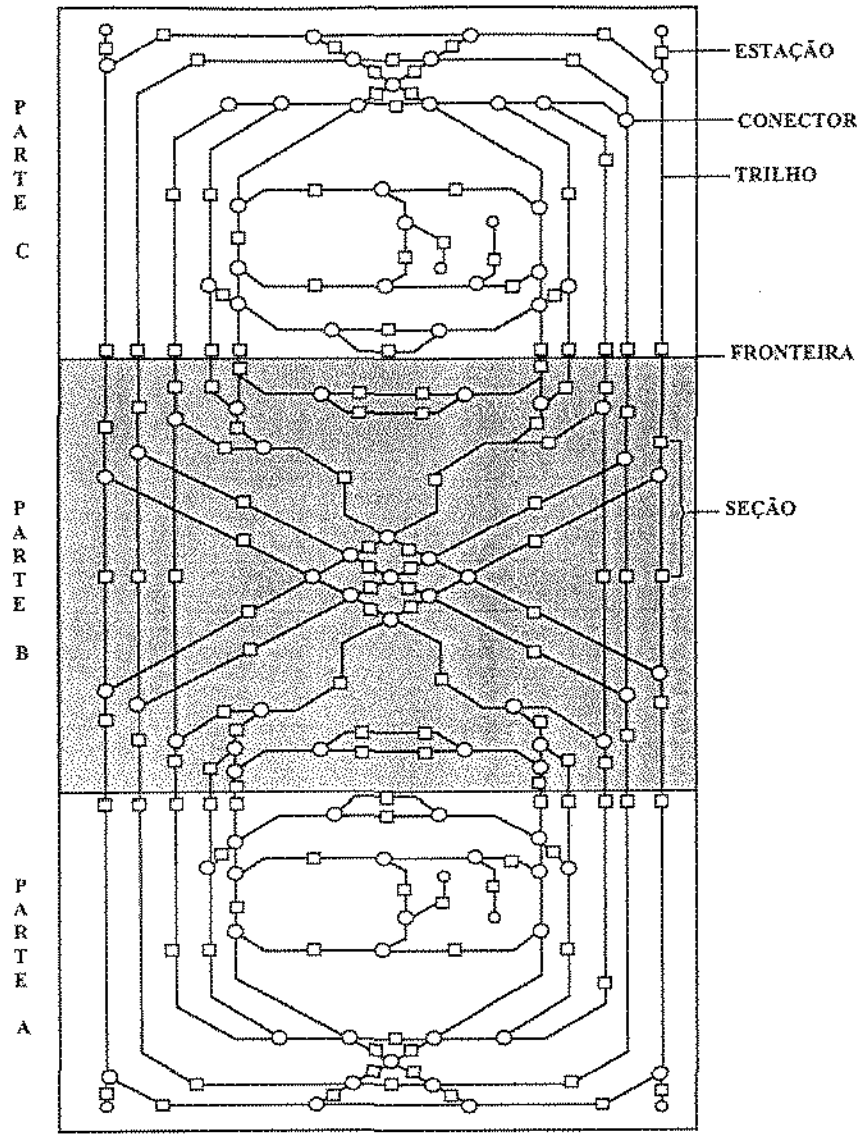


Figura 29 - O Leiaute da Malha Ferroviária

Em síntese, podemos apresentar as seguintes características e restrições para o modelo de ferrovia:

- o principal objetivo é garantir que não haja colisões entre trens;
- descarrilhamento de trens não é considerado;
- conectores podem sofrer falhas de ambiente;
- sensores também podem sofrer falhas de ambiente, mas considerando dois sensores consecutivos, supõe-se que apenas um deles pode falhar;
- o trajeto de trens, bem como a detecção de bloqueio (*deadlock*) não são

tratados;

- trens podem parar dentro de uma seção da malha;
- o comprimento de um trem é menor do que a menor seção da malha inteira, ou seja, um trem está completamente contido dentro de qualquer seção da malha;
- o sistema é responsável por detectar quando um trem irá atravessar a fronteira entre duas partes quaisquer da malha;

A definição da localização inicial dos trens, do percurso que ele deve seguir inicialmente e das precauções a serem tomadas contra colisões é responsabilidade do operador do sistema;

4.2 Requisitos Funcionais

Uma técnica bastante utilizada para extrair as classes de objetos que irão compor o sistema consiste em identificar estas classes a partir do texto da especificação do sistema (metodologia OMT [Rum92]). Primeiramente, classes candidatas são identificadas sem restrições; geralmente correspondem aos substantivos do texto. Em seguida, classes incorretas e classes desnecessárias são descartadas, de acordo com alguns critérios, tais como: classes redundantes, classes irrelevantes, atributos, operações, etc. Porém, é muito difícil identificar todos os objetos que irão compor o sistema diretamente da especificação inicial do mesmo [Jac92]; um melhor entendimento do sistema é necessário antes que os objetos possam ser identificados. Para tanto, podemos especificar os requisitos funcionais, ou seja, delimitar o sistema e definir a funcionalidade que ele pode oferecer. Seguindo a metodologia UML, utilizamos o Diagrama de Casos de Uso para representar esses requisitos. Um Diagrama de Casos de Uso é composto de **atores**, que representam os elementos que interagem com o sistema e **casos de uso**, que representam "o que" estes elementos podem fazer com o sistema. Cada caso de uso representa um fluxo completo de eventos no sistema, do ponto de vista dos elementos que interagem com o mesmo.

A Figura 30 apresenta os atores que interagem com o Controlador de Trens. O Operador utiliza o Controlador para gerenciar e monitorar o ambiente. Os Sensores monitoram as posições dos trens na malha e seus sinais são lidos pelo Controlador. Os Conectores recebem comandos do Controlador para efetuar roteamentos durante o trajeto

de um trem. Os Trens, por sua vez, interagem com o Controlador para definição de posição e direção, entre outros. E temos também a representação para a Rede de Computadores, que o Controlador utiliza para travessia dos trens pela fronteira.

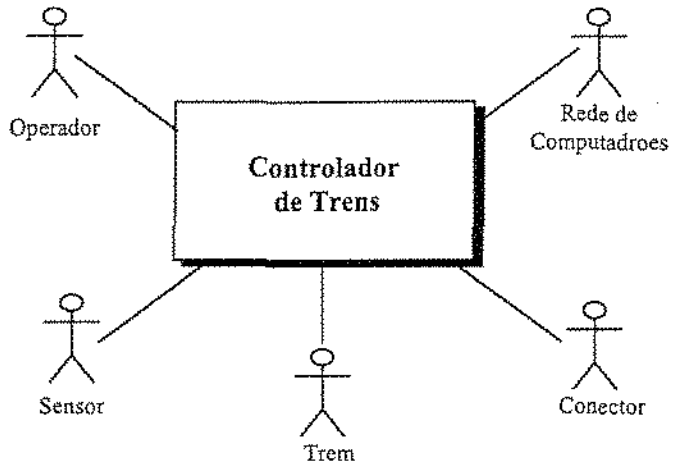


Figura 30 - Os Atores que interagem com o Controlador de Trens

Com base na investigação da interação dos atores com o sistema, podemos identificar os seguintes casos de usos (que representam a funcionalidade principal do sistema):

(1º) Gerência de requisitos do ambiente

a) O operador inicia o sistema;

b) A estrutura abstrata da malha ferroviária é montada, contendo todas as informações do leiaute físico (todas as possibilidades de movimentação dos trens);

c) O operador define, com base na estrutura da malha ferroviária, um possível trajeto para os trens a partir de determinada posição inicial. Porém, os trens percorrem a malha aleatoriamente durante a execução do sistema, de acordo com a disponibilidade das seções da malha;

d) O operador informa ao sistema qual a posição inicial do trem, isto é, a seção da malha onde ele iniciará o movimento;

(2º) Monitoração da seqüência de eventos do ambiente

a) O operador utiliza a visão da malha ferroviária para acompanhar a posição atualizada dos trens;

b) O operador monitora a ativação/desativação de sensores e a orientação dos conectores;

Fluxos Alternativos:

(i) um sensor é inesperadamente ativado devido a algum defeito ocorrido no ambiente ou quando uma ativação anterior esperada não ocorre;

(ii) Um conector falha;

(iii) Estas duas situações de falhas de ambiente são toleradas pelo sistema. Porém, a recuperação dos componentes falhos depende de intervenção manual. Neste caso, para simplificar, supomos que esta intervenção é feita pelo operador;

(3º) Movimentação do trem na malha ferroviária

a) O sistema reserva as seções necessárias para iniciar o movimento do trem;

b) O trem é inserido na posição inicial;

c) Uma próxima seção é reservada;

d) O trem ocupa a próxima seção do seu trajeto;

e) O trem libera a seção anterior que estava reservada;

Fluxos Alternativos:

(i) Quando a próxima seção não pode ser reservada, o trem tenta uma outra seção. Caso não haja seção desocupada, o trem pára e aguarda a desocupação de uma seção;

(ii) Quando um trem encontra um final de linha, ele pára e reverte a posição, podendo assim continuar o percurso;

(iii) O sistema é responsável por evitar colisões;

(4º) Travessia do trem pela fronteira

a) Quando o trem vai atravessar uma fronteira, o controlador da parte A da malha requisita à rede a reserva da seção remota (pertencente à parte B da malha);

b) O controlador da parte B reserva a seção;

c) A rede informa ao controlador da parte A que a reserva foi realizada;

d) O controlador da parte A remove o trem e solicita à rede a inserção do trem na seção remota;

e) O controlador da parte B ativa a posição inicial do trem e este inicia o movimento;

f) A rede solicita ao controlador da parte A que libere a seção que estava sendo utilizada pelo trem;

g) O controlador da parte A libera a seção;

Fluxos Alternativos:

(i) Não é possível reservar uma seção remota;

(ii) Falha nos componentes da rede inviabiliza a execução do protocolo de comunicação;

(5º) Monitoração de sensores

a) O sensor permanece desativado;

b) Quando um trem que está ocupando uma seção adjacente ao sensor move para a outra seção adjacente ao sensor, este é ativado, indicando que o trem passou sobre ele;

c) O sensor é desativado depois que o sistema ler o seu valor;

Fluxo Alternativo:

(i) O sensor falha;

(6º) Roteamento de Trens

a) Ao reservar uma seção, o sistema reserva todos os conectores contidos nesta seção e os posiciona para a orientação reto ou curvo;

b) O trem entra na seção e move-se conforme a orientação dos conectores;

c) Ao liberar uma seção, todos os conectores desta seção são liberados;

d) Os conectores continuam com a última configuração até que uma nova reserva seja realizada;

Fluxo Alternativo:

(i) O conector falha;

A Figura 31 apresenta o diagrama dos principais Casos de Uso do Controlador de Trens. As setas tracejadas representam relacionamentos de dependência entre os casos de uso.

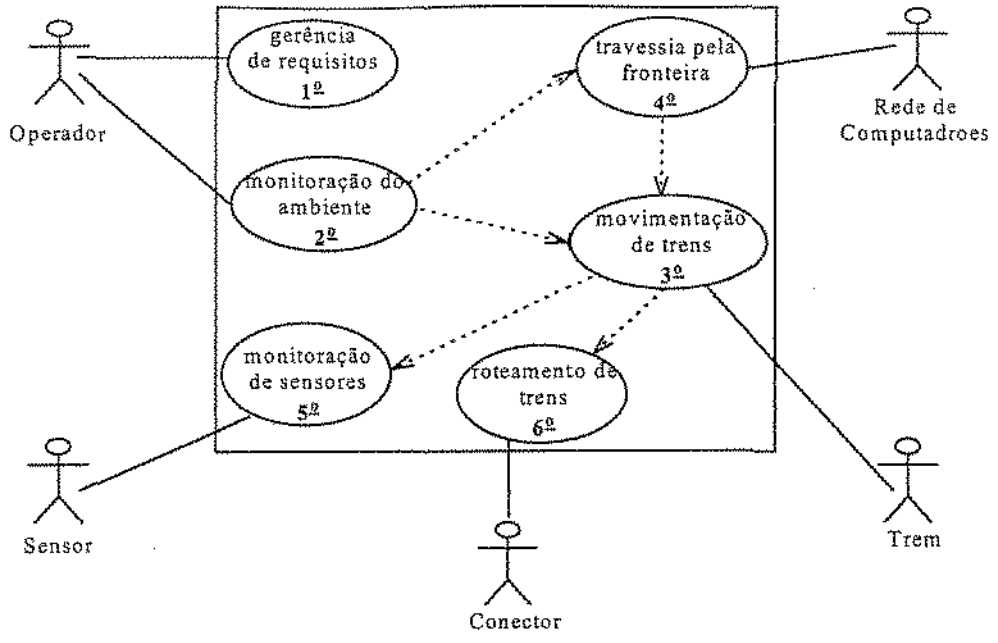


Figura 31 - Diagrama de Casos de Uso do Controlador de Trens

4.3 Modelo de Análise

O Modelo de Análise[3] concentra-se na estrutura lógica do sistema independente do ambiente de implementação. O objetivo é definir uma estrutura robusta, estável e de fácil manutenção e extensão.

Os principais recursos da UML utilizados no nosso Modelo de Análise são: o Diagrama de Classes, que descreve a estrutura estática do sistema, com base nos conceitos de classes, objetos, módulos e relacionamentos e o Diagrama de Colaborações, que enfatiza o comportamento dinâmico dos objetos através do fluxo de operações (seqüenciais ou concorrentes).

4.3.1 Diagrama de Classes

O Diagrama de Classes foi construído com base nos casos de uso identificados. Porém, a UML não prescreve um mapeamento particular entre os casos de usos e a estrutura de

[3] O termo "Modelo de Análise" foi extraído da metodologia OOSE [Jac92], que também faz parte da unificação que gerou a UML.

classes do modelo. Uma abordagem interessante (proposta por Jacobson [Jac92]) consiste em identificar primeiramente as classes de objetos da interface, que modelam os elementos que interagem com o sistema; em seguida, as classes de objetos de armazenamento, que modelam as informações que o sistema mantém durante a execução; e, por fim, as classes de objetos de controle, que modelam a funcionalidade não extraída pelos outros objetos como, por exemplo, o comportamento resultante de operações com os objetos de armazenamento, que retornam o resultado para os objetos da interface. A principal razão para modelar usando estas três categorias de objetos é facilitar futuras mudanças no sistema, ou seja, as mudanças em alguma parte da interface ou em alguma estrutura de armazenamento ou mesmo em alguma parte do controle irão afetar apenas um determinado conjunto de classes.

Para identificar as classes de objetos da interface nós nos concentramos na interação dos atores com o sistema. Analisando o nosso Diagrama de Casos de Uso (Figura 31) identificamos duas importante classes de interface com o Operador: A classe Interface do Operador e a classe Visão da Malha. Através da classe Interface do Operador é possível criar trens, especificar a localização inicial e o trajeto dos trens, determinar as precauções a serem tomadas para evitar colisões, determinar quais as concessões para possíveis falhas de ambiente e determinar quando iniciar e parar o sistema. A classe Visão da Malha é utilizada pelo operador para a monitoração do sistema, ela contém operações para mostrar o comportamento do trem na malha ferroviária e toda informação relevante como, por exemplo, estados dos sensores e conectores. Os atores Trem, Sensor e Conector fazem parte do hardware da ferrovia. Portanto, definimos uma classe Hardware da Ferrovia, que modela a interação do sistema com trens, sensores e conectores. Esta classe fornece operações para ajustar a velocidade do trem, mudar a direção do trem, ler indicações de sensores, etc. Para o ator Rede de Computadores, definimos a classe Protocolo de Comunicação, que representa a interface com a rede de computadores. Esta classe encapsula as operações básicas de ativação dos componentes da rede e fornece operações para a comunicação entre os objetos distribuídos do sistema.

As classes de objetos de armazenamento são, na maioria das vezes, extraídas diretamente da descrição dos casos de usos. Em nosso caso, o principal componente de armazenamento decorre do passo (b) do (1º) caso de uso "Gerência de requisitos do

ambiente" (Seção 4.2). Este passo faz referência à estrutura da malha ferroviária, que é definida na classe Malha. A classe Malha generaliza uma série de componentes — sensores, estações e conectores — que serão detalhados mais adiante.

A principal classe de objeto de controle, denominada classe Controle Central, constitui a parte central do sistema e é responsável pelo controle dos diversos componentes da ferrovia: malha, visão da malha, trens e hardware da ferrovia. Além disso, objetos da classe Controle Central comunicam-se com a Interface do Operador, informando a posição atual do trem e também, informações de caráter excepcional como, por exemplo, quando o disparo de um sensor ocorre inesperadamente ou quando um disparo esperado não ocorre. Para o caso do controle de trens, definimos a classe Trem, que fornece operações para controlar a movimentação dos trens sobre a malha. Um objeto da classe Trem envia mensagens do tipo ReservaSeção e LiberaSeção, as quais são definidas na classe Controle Central. Este objeto também envia mensagens do tipo AtualizaPosiçãoTrem, que é definida na classe Visão da Malha e AjustaVelocidade, que é definida na classe Hardware da Ferrovia.

A Figura 32 apresenta os principais componentes do diagrama de classes do Controlador de Trens. A classe Controle Central pode enviar mensagens para a classe Interface do Operador e a classe Interface do Operador pode enviar mensagens para a classe Controle Central. Cada instância da classe Controle Central encapsula uma interface para o hardware da ferrovia, uma parte da malha ferroviária, a visualização desta malha e os trens. Mais especificamente, a classe Controle Central é uma agregação contendo uma instância da classe Hardware da Ferrovia, uma instância da classe Malha, uma instância da classe Visão da Malha e várias instâncias da classe Trem. Similarmente, cada instância de Hardware da Ferrovia, Trem, Malha e Visão da Malha pertence ao mesmo Controle Central. Objetos da classe Controle Central podem enviar mensagem para objetos das classes Hardware da Ferrovia, Trem, Malha e Visão da Malha e estes, por sua vez, podem enviar mensagens para a instância da classe Controle Central. Além disso, cada par de classes Visão da Malha e Trem, Trem e Hardware da Ferrovia, Malha e Hardware da Ferrovia relacionam-se em ambas as direções.

Uma instância da classe Interface do Operador está associada a uma ou mais instâncias da classe Controle Central. Para o nosso protótipo em particular, consideramos uma

instância da classe Interface do Operador e três instâncias da classe Controle Central. Cada instância da classe Controle Central usa uma instância da classe Protocolo de Comunicação para se comunicar com outro objeto Controle Central distribuído.

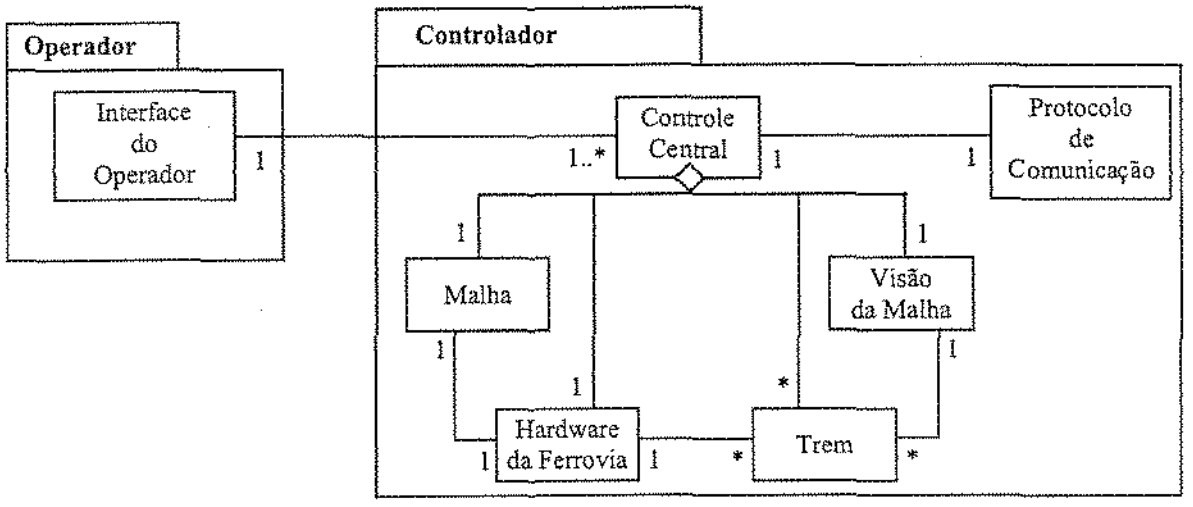


Figura 32 - Principais Componentes do Diagrama de Classes

Para maior clareza, organizamos os componentes do Controlador de Trens em dois módulos principais: O módulo Operador e o módulo Controlador. Cada módulo Controlador controla uma parte da malha ferroviária e corresponde, em tempo de execução, a um componente distribuído na rede de computadores. Os controladores comunicam-se entre si e também com o operador (Figura 33).

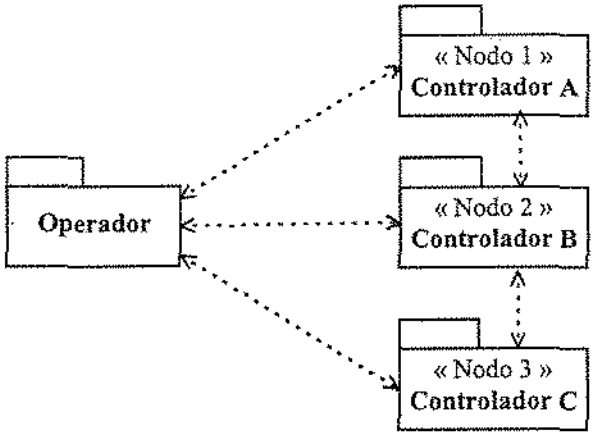


Figura 33 - Dependências entre o Operador e os Controladores Distribuídos

A seguir descrevemos em detalhes os componentes Trem, Malha e Protocolo de Comunicação. A descrição destes componentes é importante para ilustrar as nossas idéias sobre construção de software orientado a objetos, atendendo aos requisitos de distribuição e confiabilidade.

4.3.1.1 Componente Trem

A Figura 34 ilustra a classe Trem. As principais operações fornecidas são Inicia e Para um trem; Move, que executa o movimento do trem para uma nova seção; e Reverte, utilizada para reverter a direção do trem.

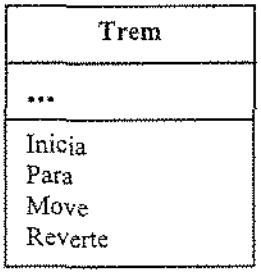


Figura 34 - Classe Trem

É importante ressaltar que o trem deve mover-se pela malha sem colisões. O conceito de região de controle é muito importante para atender este requisito. Uma **região de controle** é um conjunto de seções predefinidas para o trajeto do trem. Cada trem é responsável pela escolha de seu trajeto na sua região de controle. Deste modo, um trem sabe qual é a próxima estação do trajeto (ou seja, o próximo sensor a ser ativado). Na Figura 35 temos um exemplo de região de controle com dois níveis: o primeiro nível contém informações sobre as próximas seções de uma determinada seção; o segundo nível contém informações das próximas seções de cada seção do primeiro nível, e assim por diante.

Quando supomos que todos os conectores e sensores da malha são elementos confiáveis, é suficiente para um trem construir uma região de controle de um nível. Além disso, não é necessário reservar todas as próximas seções da seção corrente; é suficiente reservar apenas uma próxima seção para garantir que não haverá colisão de trens. Entretanto, se considerarmos a possibilidade de ocorrerem falhas em sensores e conectores,

o trem precisa reservar um número maior de seções da região de controle. Nós comentaremos estas e outras questões sobre tolerância a falhas de ambiente na Seção 4.4.

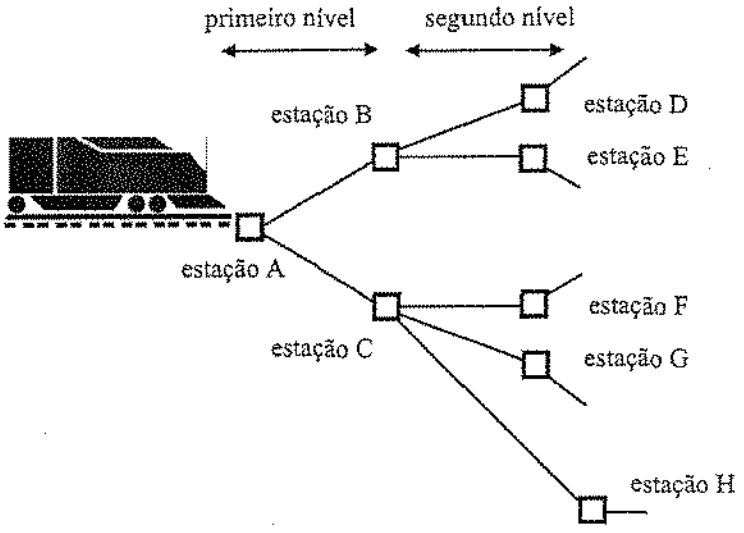


Figura 35 - Exemplo de Região de Controle com Dois Níveis

4.3.1.2 Componente Malha

A classe Malha é composta por um conjunto de seções, as quais são compostas por estações (isto é, sensores) e conectores (Figura 36).

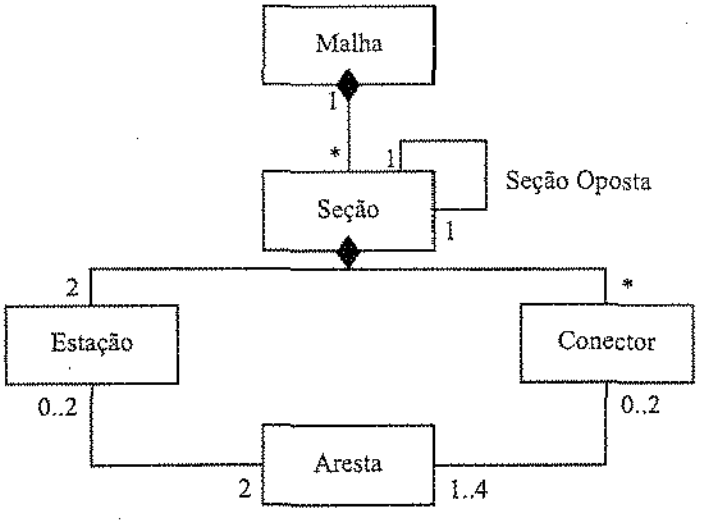


Figura 36 - Componente Malha

Como mencionado anteriormente, uma **seção** é uma ligação orientada entre duas estações adjacentes. Cada seção, delimitada por duas estações adjacentes, pode conter uma seqüência de zero ou mais conectores (Figura 37). Conectores e estações são ligados entre si através de **arestas**, que são partes do trilho da malha. A direção de uma seção S é definida da estação cauda t para a estação cabeça h. Por exemplo, na Figura 37 a seção definida por duas estações adjacentes A e B é diferente da seção que liga B com A. Assim, uma seção é associada com sua seção oposta. A associação um-para-um Seção Oposta na Figura 36 mostra este relacionamento.

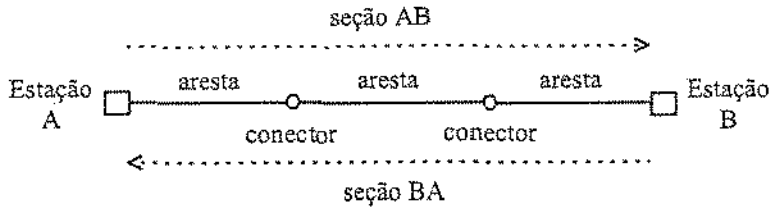


Figura 37 - Exemplo de Seção

Existem três tipos de conectores na malha ferroviária: travessia, final de linha e desvio. Um conector travessia (Figura 38d) é um tipo estático de conector, que não muda de direção. Um conector final de linha (Figura 38c) é um conector terminal na malha. Um conector de desvio é um conector que tem duas direções controláveis: reto e curvo. Existem dois tipos de conectores de desvio: bifurcação (Figura 38a) e cruzamento (Figura 38b). Um conector bifurcação está associado a três arestas, enquanto um conector cruzamento está associado a quatro arestas. Um conector travessia está associado também a quatro arestas e um conector final de linha está associado com apenas uma aresta. A Figura 36 mostra a associação entre as classes Conector e Aresta. Dependendo do tipo de conector, ele pode estar associado a uma ou até quatro arestas, inversamente, cada aresta pode estar associada a zero ou até dois conectores.

Componente Seção

O controle central é responsável por informar qual é a próxima seção de uma determinada seção. As próximas seções de uma determinada seção representam uma importante propriedade da estrutura do Controlador de Trens. Dada uma seção S com um estação cauda t e uma estação cabeça h, as próximas seções de S são as seções cuja estação cauda é

h. Por exemplo, a Figura 39 ilustra uma parte da malha, a qual contém quatro estações — estação A, estação B, estação C e estação D, um conector cruzamento nomeado cruzamento1, e dois conectores bifurcação nomeados bifurcação2 e bifurcação3. Assim, podemos identificar as seções descritas na Tabela 1. Por exemplo, as próximas seções da seção ocupada pelo trem (cuja estação cabeça é a estação A) são seção 1, seção 3 e seção 5. As seções opostas de seção 1, seção 3 e seção 5 são, respectivamente, seção 2, seção 4 e seção 6.

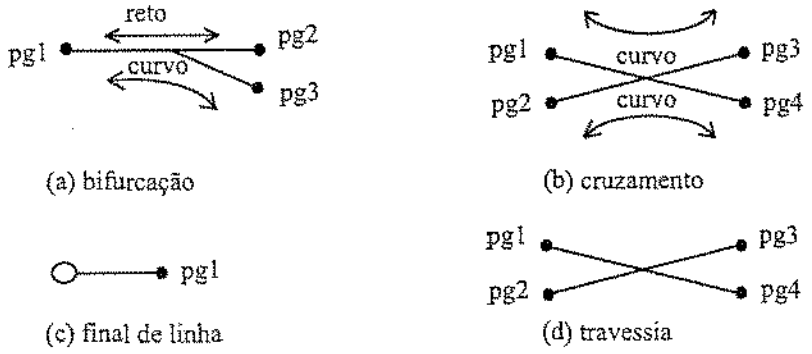


Figura 38 - Tipos de Conectores

Seção	Cauda	Cabeça	Caminho
seção 1	estação A	estação B	cruzamento1 (reto), bifurcação3 (reto)
seção 2	estação B	estação A	bifurcação3 (reto), cruzamento1 (reto)
seção 3	estação A	estação D	cruzamento1 (curvo), bifurcação2 (curvo)
seção 4	estação D	estação A	bifurcação2 (curvo), cruzamento1 (curvo)
seção 5	estação A	estação C	cruzamento1 (curvo), bifurcação2 (reto)
seção 6	estação C	estação A	bifurcação2 (reto), cruzamento1 (curvo)

Tabela 1 - Exemplos de Seções

Nós podemos identificar três diferentes tipos de seções na malha: seção sólida, seção particionada e seção interconectada. Uma *seção sólida* tem próxima seção enquanto uma *seção particionada* não tem uma próxima seção. A Figura 40 mostra um exemplo de uma típica seção particionada, que pode ser encontrada no leiaute do Controlador de trens. Como conseqüência, quando o trem ocupa uma seção particionada, o controle central é responsável por parar e reverter o trem antes de encontrar o final da linha. Uma *seção interconectada* se encontra na fronteira de duas partes da malha, ou seja, sua próxima seção

é controlada por um controle central remoto. Neste caso, o controle central da parte em que começa a seção interconectada é responsável por comunicar-se com o controle central da parte em que termina a seção, para estabelecer um protocolo de transferência de controle do trem entre as partes da malha.

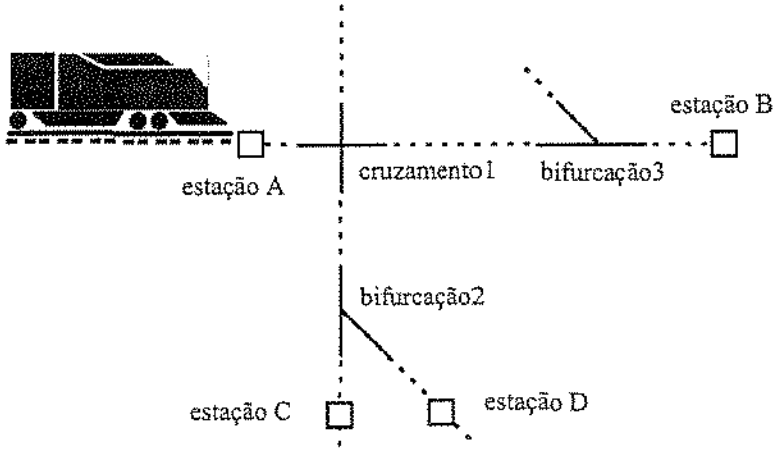


Figura 39 - Exemplo de Próximas Seções de uma Seção

Supondo que o comportamento desses três tipos de seções é estático, ou seja, conhecemos antecipadamente quais as seções que são sólidas, particionadas ou interconectadas, podemos criar uma hierarquia de classes, onde três diferentes classes derivadas (Seção Sólida, Seção Particionada e Seção Interconectada) herdam as características comuns de uma classe base Seção (Figura 41). Os métodos Reserva e Libera implementam a funcionalidade básica de reserva e liberação de seções sólidas ou particionadas e são sobrepostos na subclasse Seção Interconectada. O método Ocupa implementa a funcionalidade básica de uma seção sólida e é sobreposto na subclasse Seção Particionada e na subclasse Seção Interconectada.

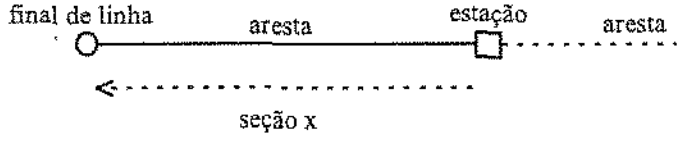


Figura 40 - Exemplo de Seção Particionada

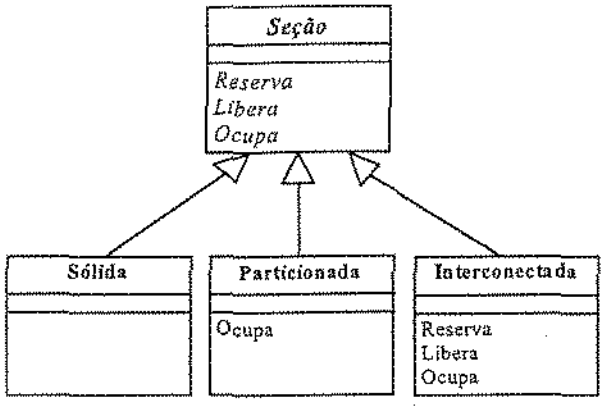


Figura 41 - Componente Seção

Componente Conector

Conectores podem ser classificados em diferentes tipos: Final de Linha, Travessia e Desvio. O tipo Desvio, por sua vez, pode ser classificado em diferentes tipos: Bifurcação e Cruzamento. A hierarquia de classes da Figura 42 encapsula os dados e comportamentos dos diversos tipos de conectores existentes na malha ferroviária (Figura 38).

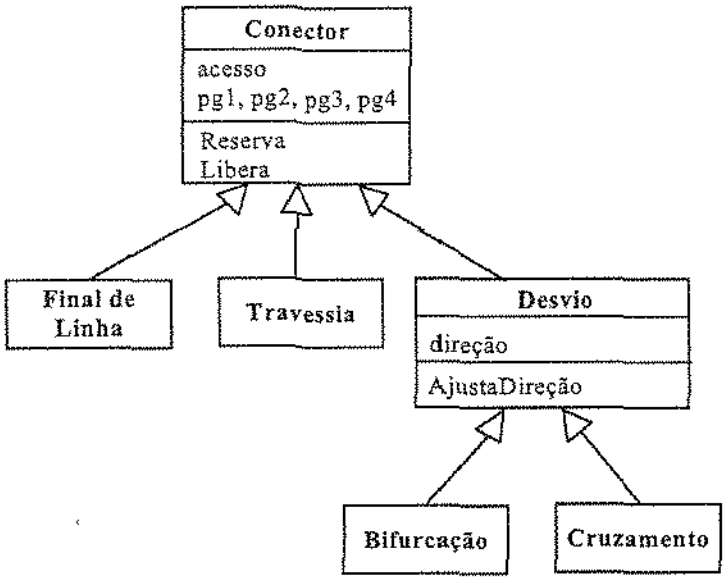


Figura 42 - Componente Conector

A classe Conector define o atributo acesso, que indica se um conector está livre ou reservado, bem como as operações Reserva e Libera, que mudam o valor do atributo

acesso. Um conector contém também pontos de guia, definidos pelos atributos pg1, pg2, pg3 e pg4, que se conectam com as arestas. Dependendo do tipo de conector, o número de pontos de guia pode ser 1 (final de linha), 3 (bifurcação) ou 4 (travessia e cruzamento), como mostrado na Figura 38. A Classe Desvio representa um tipo de conector que tem uma direção reta ou curva, como especificado pelo atributo direção, bem como uma operação AjustaDireção utilizada para mudar o valor do atributo direção.

4.3.1.3 Componente Protocolo de Comunicação

O ambiente alvo para o desenvolvimento do Controlador de Trens é um sistema distribuído contendo múltiplas estações SPARC conectadas através de uma rede TCP/IP (Figura 43). Este sistema é organizado através do modelo cliente-servidor.

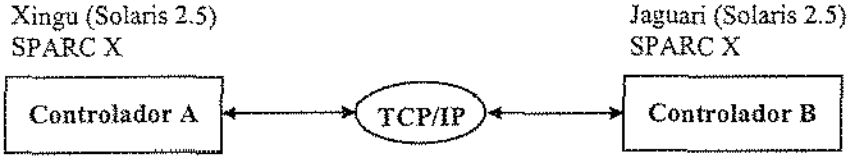


Figura 43 - Ambiente Alvo

A classe ProtocoloComunicação (Figura 44) implementa a funcionalidade básica para comunicação entre dois controladores distribuídos e, portanto, encapsula detalhes de comunicação entre processos, tais como, estabelecimento da comunicação, empacotamento de argumentos e desempacotamento de resultados. Esta classe mantém um apontador (host) para a máquina onde se encontra o controlador remoto. As principais operações disponíveis na interface da classe ProtocoloComunicação são: SolicitaReserva, utilizada para solicitar a um controlador remoto, a reserva de uma seção; SolicitaLiberação, utilizada para solicitar a um controlador remoto, a liberação de uma seção; InsereTrem, utilizada para solicitar a um controlador remoto, a inserção de um trem na malha ferroviária; e, RemoveTrem, utilizada para solicitar a um controlador remoto, a retirada de um trem da malha ferroviária.

A classe ProtocoloComunicação foi implementada utilizando-se mecanismos de chamada de procedimento remoto[4] e, portanto, fornece uma interface transparente para comunicação entre os controladores. Como vimos no Capítulo 3 (Seção 3.2.2.2), *RPC*

[4] Protocolo SunRPC, Solaris 2.5.

consiste de uma abstração para a comunicação cliente-servidor, com o objetivo de esconder as camadas da rede, de forma que pareça que estamos usando uma interface convencional de chamada de procedimento.

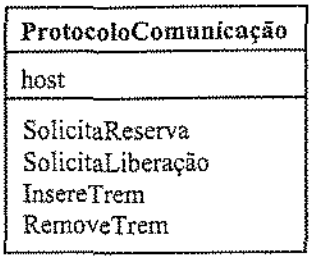


Figura 44 - Componente Protocolo de Comunicação

A principal vantagem do componente Protocolo de Comunicação é que ele separa os serviços de comunicação da funcionalidade básica do Controlador de Trens. Desta forma, podemos alterar os mecanismos de comunicação da rede, sem afetar a funcionalidade do restante do sistema. Podemos, por exemplo, acrescentar suporte para tolerância a falhas de comunicação, alterando exclusivamente a implementação da classe ProtocoloComunicação, sem que seja necessário alterar a sua interface.

Na Seção 4.3.2.2, apresentamos um diagrama de colaborações, que descreve o comportamento do componente Protocolo de Comunicação e na Seção 4.5.1 apresentamos alguns aspectos de implementação deste protocolo.

4.3.2 Diagrama de Colaborações

A seguir apresentamos os diagramas de colaborações para dois importantes fluxos de operações do Controlador de Trens: (1) movimentação do trem dentro de uma parte da malha e (2) travessia do trem pela fronteira. O contexto de um diagrama de colaborações mostra os objetos relevantes para a realização de uma determinada operação, incluindo objetos indiretamente afetados ou acessados durante a operação [BRJ97].

As mensagens que implementam uma operação são numeradas inicialmente com o número 1. Para um fluxo de controle interdependente, os números das mensagens subseqüentes são aninhados de acordo com o aninhamento das chamadas (por exemplo, 1,

1.1, 1.1.1). Para uma seqüência de mensagens independentes, os números pertencem ao mesmo nível, isto é, não são aninhados (por exemplo, 1.1, 1.2, 1.3).

4.3.2.1 Movimentação do Trem dentro de uma parte da Malha

A Figura 45 apresenta um diagrama de colaborações contendo o comportamento dinâmico das operações realizadas para a movimentação do trem dentro de uma parte da malha. As operações ocorrem, em termos gerais, da seguinte maneira: primeiro, o Controle Central ativa a movimentação do Trem; o Trem, por sua vez, solicita ao Controle Central a reserva da seção para a qual ele vai mover; o Controle Central solicita a reserva ao objeto Seção; o objeto Seção solicita ao objeto Conector, a reserva de todos os conectores (1..n) pertencentes à seção. Depois de reservada a seção, o Trem solicita ao Controle Central a ocupação desta seção e, após uma nova movimentação, solicita ao Controle Central a liberação da mesma. Quando um Trem está realizando a movimentação na malha, ele solicita à Visão da Malha a atualização da posição e, eventualmente, solicita ao Hardware da Ferrovia, o ajuste da velocidade.

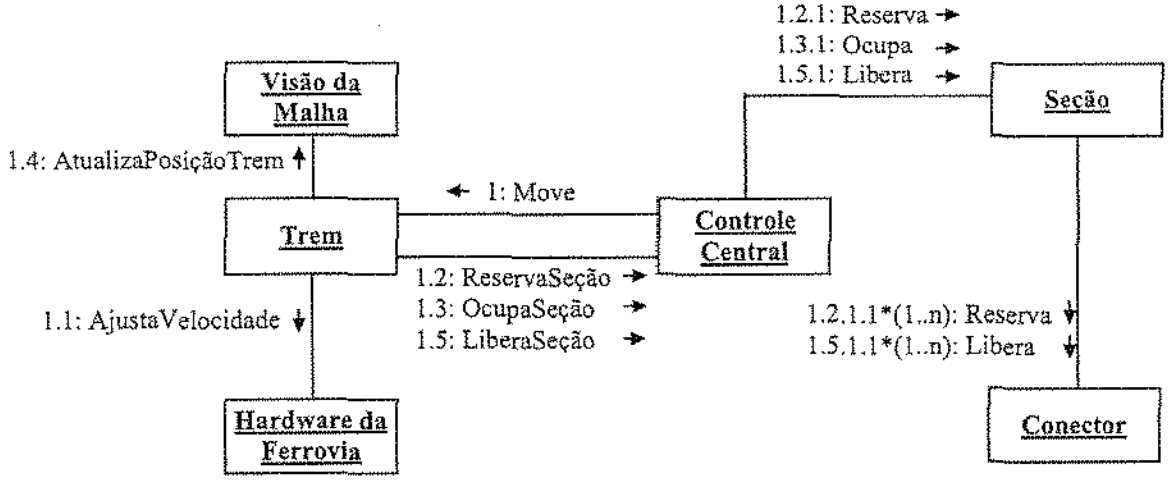


Figura 45 - Movimentação do Trem dentro de uma Parte da Malha

4.3.2.2 Travessia do Trem pela Fronteira

A Figura 46 apresenta um diagrama de colaborações contendo o comportamento dinâmico das operações realizadas quando um trem move para uma seção interconectada. Este

movimento para uma seção interconectada ocorre, em termos gerais, da seguinte maneira: primeiro, a reserva da seção é realizada normalmente; em seguida, o objeto Seção envia uma mensagem ao Controle Central A solicitando a reserva do complemento da seção, que está sob controle remoto; o Controle Central A solicita, através do Protocolo de Comunicação, a reserva do complemento da seção. Se a reserva requerida for atendida então o Controle Central A solicita, através do Protocolo de Comunicação, a inserção do trem na outra parte da malha. O Controle Central B solicita ao Controle Central A, a remoção do trem e a liberação da seção que estava reservada, ao passo que insere o trem na nova parte da malha e assume o controle do mesmo, iniciando o seu percurso.

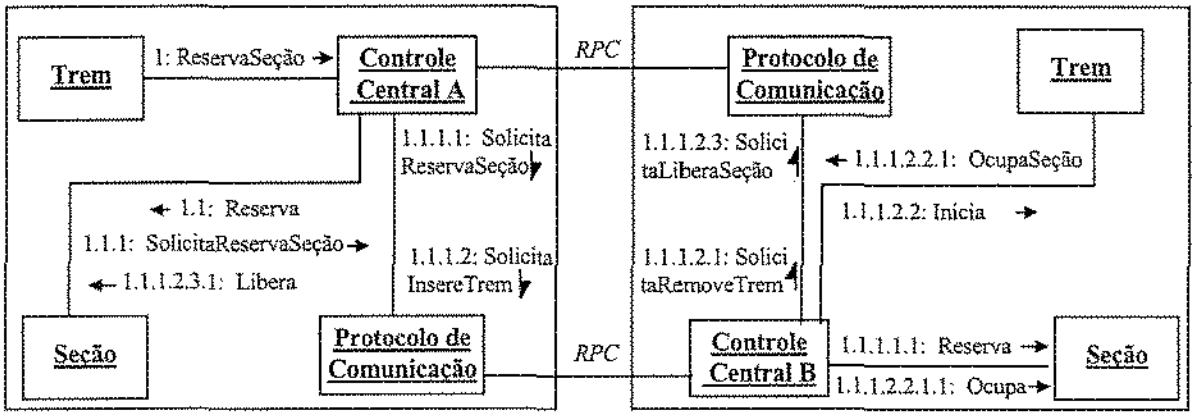


Figura 46 - Travessia do Trem pela Fronteira

4.4 Tolerando Falhas de Ambiente

Sensores e conectores de desvio são as fontes de falhas de ambiente que podem ocorrer no sistema. A seguir apresentamos os tipos de falhas considerados e os mecanismos utilizados para tolerá-las.

4.4.1 Falhas de Conectores

A ocorrência de uma falha eletro-mecânica em um conector de desvio da malha ferroviária muda sua fase de comportamento. O mecanismo de hierarquia de estados e delegação pode ser utilizado para modelar este requisito, como mostra a Figura 47.

Um conector de desvio pode estar no estado normal ou no estado anormal. Um conector de desvio anormal pode sofrer três tipos de falhas: sempre reto, quando tem a orientação fixa na posição reto; sempre curvo, quando tem a orientação fixa na posição curvo; e, indefinido, que representa qualquer outro estado anormal não identificado previamente.

A implementação da classe Desvio precisa ser alterada, mas sua interface permanece a mesma. Primeiro, nós criamos uma nova hierarquia EstadoDesvio para modelar os diferentes estados de um conector de desvio. Em seguida, delegamos o método AjustaDireção para uma variação apropriada na hierarquia. Desta forma, mantemos as mesmas abstrações e mecanismos do nosso modelo; apenas adicionamos os estados do comportamento anormal de maneira incremental.

Na Figura 47, temos uma classe Desvio, que delega seu comportamento conforme o estado do conector de desvio. A classe de um objeto Desvio muda quando seu estado muda, efetivamente mudando o objeto delegado. A classe DesvioNormal implementa o serviço normal da operação AjustaDireção, ao passo que a classe DesvioAnormal implementa o serviço anormal desta operação. As classes derivadas de DesvioAnormal devem reimplementar esta operação conforme o tipo de falha.

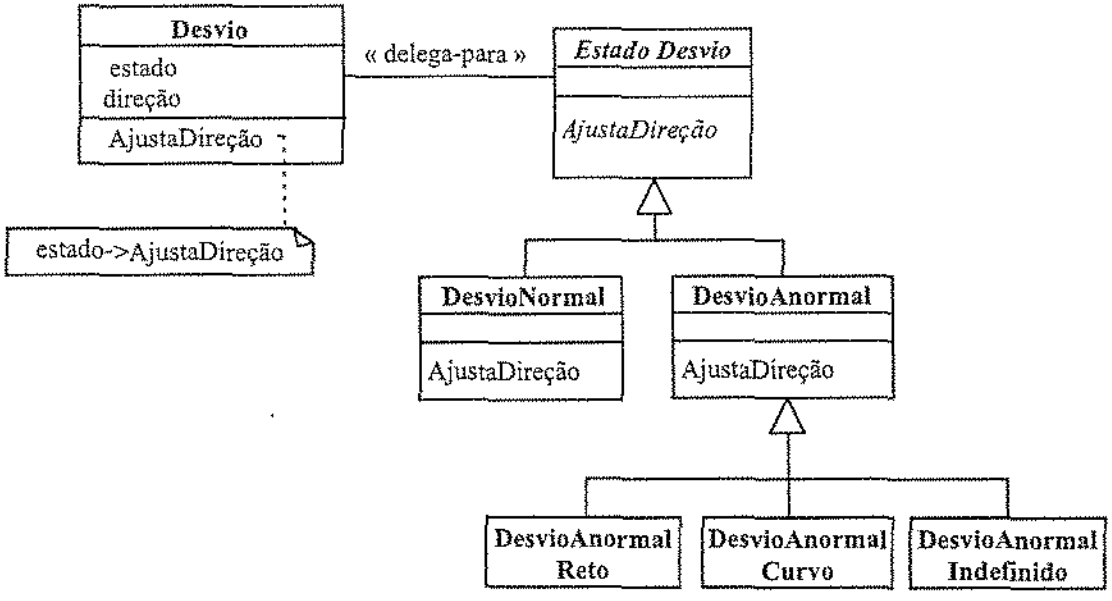


Figura 47 - Hierarquia de Estados para o Conector de Desvio

4.4.2 Falhas de Sensores

Um sensor representa um estado normal quando ele resulta em um sinal correto na detecção de trens, ou seja, resulta em sinal "1" se um trem passa sobre o sensor e sinal "0" quando no modo permanente de leitura; caso contrário, supõe-se que o sensor sofreu uma falha.

O ponto inicial para toda estratégia de tolerância a falhas é a detecção de um estado errôneo [LA90]. No caso do Controlador de Trens, um trem é capaz de detectar um erro na sua posição corrente baseado num mecanismo de tratamento de exceções. Como vimos, cada trem é responsável pela escolha de seu trajeto na sua região de controle. Deste modo, um trem sabe qual é a próxima estação do trajeto (ou seja, o próximo sensor a ser ativado). Se um sensor é ativado inesperadamente fora da sua região de controle, o trem sinaliza uma exceção para o controlador. Após a detecção de um erro é necessário tolerá-lo por meio de técnicas de recuperação de erros. Em nossa aplicação, utilizamos recuperação de erros por avanço: quando um trem detecta que seu trajeto foi desviado, ele tenta ativar um novo trajeto na sua região de controle, visando recuperar-se do erro.

4.4.3 Estendendo a Classe Trem

Quando supomos que todos os conectores de desvio e sensores da malha são elementos confiáveis, é suficiente para um trem reservar apenas uma próxima seção de sua região de controle, para garantir que não haverá colisão de trens. Entretanto, se supusermos que conectores de desvio podem falhar, um trem precisa reservar todas as próximas seções de sua seção corrente. Dessa forma, quando um conector de desvio falha, o trem pode desviar do seu trajeto original, pois todas as seções seguintes foram reservadas para ele. Conseqüentemente, a noção de região de controle é de fundamental importância para o algoritmo de recuperação do trem. Na Figura 48, criamos uma nova classe, TremRobustoCon, para modelar um trem que pode detectar e recuperar-se de falhas em conectores de desvio. A classe TremRobustoCon herda da classe Trem e redefine o método Move, para dar suporte à nova representação da região de controle. A classe Trem permanece inalterada. Ainda na Figura 48, apresentamos uma nova classe, TremRobustoConSen, para modelar um trem que move pela malha, detectando e recuperando-se de falhas em conectores e sensores. A classe TremRobustoConSen herda

de TremRobustoCon, que por sua vez herda de Trem. A classe TremRobustoCon permanece inalterada.

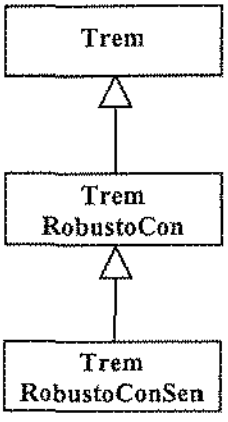


Figura 48 - Componente Trem Tolerante a Falhas de Conectores e Sensores

A classe TremRobustoConSen redefine o método Move de TremRobustoCon para implementar uma região de controle com três níveis ao invés de dois. Quando supomos que conectores e sensores podem falhar, nós precisamos de uma região de controle maior (composta por mais níveis) do que a considerada para falhas de conectores. Devido ao fato de termos suposto que não ocorre falha em dois sensores consecutivos da malha, um trem tolerante a falhas de conectores e sensores precisa construir uma região de controle com apenas um nível a mais do que um trem tolerante a falhas apenas de conectores.

4.4.4 Estendendo a Classe Seção

A classe Seção pode ser estendida para que o Controle Central possa informar o mais rápido possível a um determinado Trem, a disponibilidade de uma seção. Para tanto, utilizamos novamente o mecanismo de hierarquia de estados e delegação. Primeiro, criamos uma nova hierarquia de estados — EstadoSeção — para modelar os estados normal e anormal de uma seção (Figura 49). Desta forma, toda vez que é detectado que um conector de desvio está falho, toda seção que possui este conector torna-se anormal.

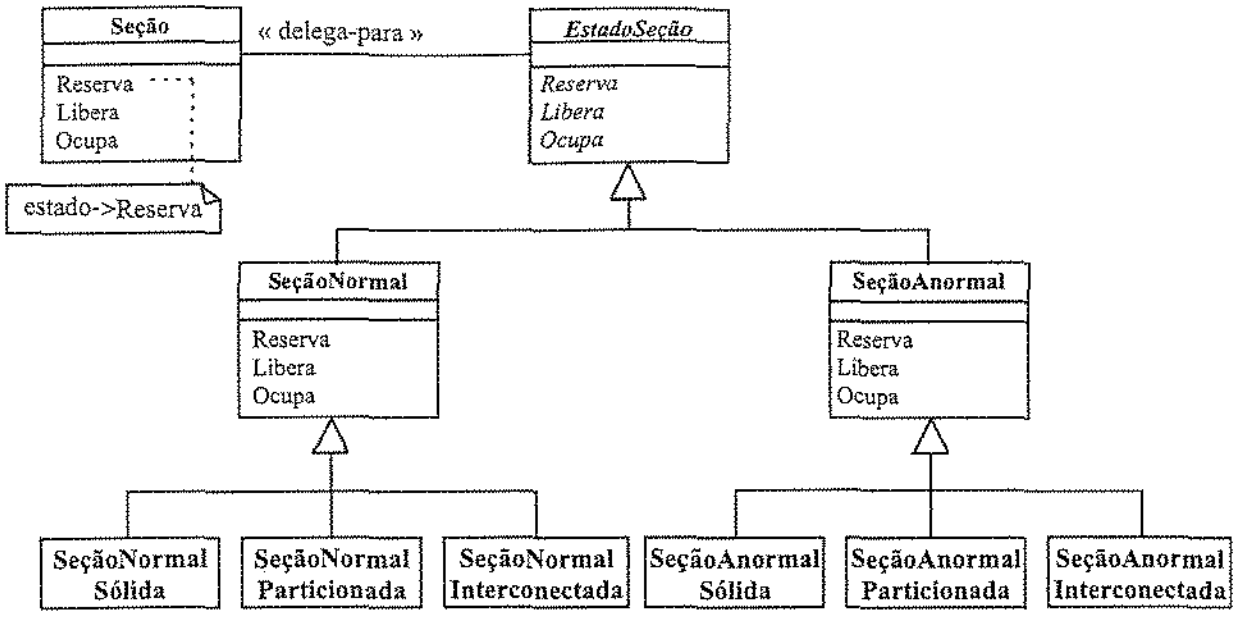


Figura 49 - Hierarquia de Estados para o Componente Seção

4.5 Aspectos de Implementação

Nesta Seção apresentamos algumas interfaces de classes (em C++) e também alguns algoritmos em pseudo-código para exemplificar a implementação de alguns aspectos do componente Protocolo de Comunicação, bem como da estrutura de tolerância a falhas em conectores. No final da Seção é feita uma breve avaliação do protótipo desenvolvido.

4.5.1 Protocolo de Comunicação

O exemplo a seguir apresenta um simples algoritmo para ilustrar o método Reserva da classe SeçãoNormalInterconectada apresentada na Figura 49. A classe SeçãoNormalInterconectada redefine o método Reserva para realizar corretamente a reserva numa seção da fronteira entre duas partes da malha. A operação reserva é realizada em dois passos: primeiro, a reserva da seção é realizada normalmente; em seguida, a reserva do complemento da seção que se encontra em outra parte da malha é requerida. Esta requisição é feita ao objeto Controle Central, que, por sua vez, utiliza os serviços do

Protocolo de Comunicação. Nas linhas 4 e 9 temos a sinalização de uma exceção para o caso de falha na tentativa de reservar uma seção.

```
1. início (método Reserva)
2.   a reserva da seção é realizada normalmente
3.   se ocorreu falha
4.      sinaliza exceção "FALHA NA OPERAÇÃO DE RESERVA"
5.      retorna
6.   fim-se
7.   Solicita ao controle central a reserva do complemento da seção
8.   se ocorreu falha
9.      sinaliza exceção "FALHA NA OPERAÇÃO DE RESERVA"
10.     retorna
11.  fim-se
12.  retorna
13.fim
```

O exemplo abaixo apresenta o código C++ para a interface da classe ProtocoloComunicação (*ControllerProtocol*) apresentada na Figura 44. O contrutor (linha 9) recebe como argumentos, as máquinas (*hosts*) onde se encontram os controladores das três partes da malha.

```
1. class ControllerProtocol
2. {
3.  private:
4.     char blueHost;
5.     char greenHost;
6.     char redHost;
7.     char currHost;
8.  public:
9.     ControllerProtocol(char* bh, char* gh, char* rh);
10.    ~ControllerProtocol();
11.
12.    void setHost(int secId);
13.    OpHistory* RequestLock(int secId, int trId, int &tarId);
14.    OpHistory* RequestLockUnconditionally(int secId, int trId, int
   &tarId);
15.    OpHistory* RequestRelease(int secId);
16.    OpHistory* InsertTrain(int trId, int front, int back, int
   origin);
17.    OpHistory* RemoveTrain(int trId);
18.};
```

Como exemplo da utilização do mecanismo de RPC na implementação dos métodos da classe ProtocoloComunicação, apresentamos a seguir o algoritmo do método SolicitaReserva. Toda chamada de procedimento remoto foi implementada de forma assíncrona, através da criação de múltiplos fluxos de execução[5] (*multi-thread*). Na linha 4 temos a sinalização de uma exceção para o caso de falhas na conexão e na linha 9 temos a sinalização de uma exceção para o caso de falhas na chamada propriamente dita.

[5] Biblioteca de *threads* — Solaris 2.5

```
1. início (método SolicitaReserva)
2.   executa a primitiva para criar um cliente antes de fazer a
   chamada
3.   se ocorreu falha
4.      sinaliza exceção "FALHA NA CRIAÇÃO DO CLIENTE RPC"
5.      retorna
6.   fim-se
7.   executa a chamada do procedimento remoto solicita_reserva
8.   se ocorreu falha
9.      sinaliza exceção "FALHA NA RPC"
10.     retorna
11.  fim-se
12.  executa a primitiva para destruir o cliente
13.  retorna
14.fim
```

4.5.2 Tolerância a Falhas de Conectores

O exemplo a seguir apresenta o código C++ para a interface (simplificada) da hierarquia de estados do conector de desvio (Figura 47). Os métodos são apresentados com base na definição de estados. O acesso ao estado corrente é realizado através das operações lerModo (*getMode*) e atribuirModo (*putMode*). A implementação da operação ajustaDireção (*directSwitch*) na classe Desvio (*Switch*) é delegada através do objeto estado (*currState*), que representa o estado corrente (linha 23).

```
1. enum SwitchSt {SW_NORMAL, SW_ABNORMAL, SW_ABNORMAL_STR,
   SW_ABNORMAL_CUR, SW_ABNORMAL_UNF};
2. enum DirectionType {STRAIGHT, CURVED, NNUULL};
3.
4. class Switch: public Connector
5. {
6. private:
7.     SwitchState         *currState;
8.     SwitchNormal        *normal;
9.     SwitchAbnormal      *abnormal;
10.    SwitchAbnormalStr   *str;
11.    SwitchAbnormalCur   *cur;
12.    SwitchAbnormalUnf   *unf;
13.    DirectionType    direction;
14.
15.    void putMode(SwitchSt m);
16.public:
17.    Switch ();
18.    ~Switch ();
19.
20.    OpHistory* directSwitch(DirectionType dir)
21.    {
22.        ...
23.        return currState->directSwitch(this,dir);
24.    }
25.
26.    void getMode(SwitchSt& m);
27.}
```

EstadoDesvio (*SwitchState*) predefine o comportamento para toda requisição delegada para ele. A interface da classe abstrata EstadoDesvio (*SwitchState*) é a seguinte:

```
1. class SwitchState
2. {
3.  public:
4.   SwitchState() { }
5.   ~SwitchState() { }
6.
7.   virtual OpHistory* directSwitch(Switch* sw, DirectionType dir) =
   0;
8.   virtual void getMode(Switch* sw, SwitchSt& m) = 0;
9. };
```

As subclasses concretas DesvioNormal e DesvioAnormal derivam de EstadoDesvio e implementam a funcionalidade da operação ajustaDireção (*directSwitch*). DesvioNormal implementa o serviço normal, enquanto DesvioAnormal implementa o serviço anormal.

4.5.3 Avaliação do Protótipo

A implementação do protótipo incorpora tolerância a falhas de conectores e sensores, bem como uma camada de comunicação, que implementa os mecanismos de RPC (Tabela 2).

Podemos afirmar que o uso de delegação como técnica de estruturação foi essencial para integrar hierarquias de classes e promover uma fácil extensão de nosso projeto.

Níveis	Número de Classes	Número de Linhas (aprox.)
Modelo Básico *	16	6.600
Camada de Comunicação	5	1.200
Tolerância a Falhas de Ambiente *	15	2.350
Total	36	10.150

* Uma primeira versão desta parte do código foi implementada em [Rub94]

Tabela 2 - Protótipo do Controlador de Trens

A implementação do protótipo final pode ainda ser melhorada, especialmente com relação a dois aspectos: (1) implementação de algoritmos para construir as regiões de controles dos trens tolerantes a falhas de sensores e conectores; (2) implementação de mecanismos confiáveis de RPC, provendo suporte para tolerância a falhas na camada de comunicação.

O código fonte do protótipo está disponível no endereço:

http://www.dcc.unicamp.br/~cmrubira/projects

4.6 Conclusões

Neste capítulo apresentamos uma experiência prática de utilização de conceitos e mecanismos de orientação a objetos para estruturar uma aplicação distribuída e tolerantes a falhas de ambiente — um Controlador de Trens.

A utilização da UML para modelar o Controlador de Trens produziu bons resultados. O diagrama de casos de uso mostrou-se bastante apropriado para a identificação dos requisitos funcionais da aplicação e, conseqüentemente, comprovou ser um importante recurso para identificação dos componentes da estrutura estática (diagrama de classes) e dinâmica (diagrama de colaborações) da aplicação. O conceito de estereótipo permitiu adicionar a semântica do mecanismo de delegação ao modelo, demonstrando ser um poderoso elemento para adicionar novos recursos de modelagem. Em síntese, a nossa experiência mostrou que a UML permite a modelagem de sistemas complexos de uma maneira bastante clara e concisa.

O requisito de distribuição foi provido de uma maneira incremental no modelo. Orientação a objetos provê computação distribuída com os mesmos benefícios que são providos para computação não distribuída: encapsulamento, reutilização, portabilidade, extensibilidade, etc. A implementação do protocolo de comunicação mostrou-se uma solução apropriada para encapsular os mecanismos de distribuição. Esta solução foi definida de maneira ortogonal e, portanto, facilita mudanças na implementação dos mecanismos de distribuição sem afetar o modelo.

Uma evolução natural para prover distribuição, em substituição aos mecanismos definidos neste trabalho, consiste na utilização de um ambiente para programação distribuída orientada a objetos (como comentado na Seção 3.2.3). Um sistema deste tipo provê suporte para programar aplicações distribuídas usando técnicas como, por exemplo, invocação de método remoto em objetos. Para tanto, tais ambiente provêem uma série de componentes para suportar invocação de métodos em objetos remotos de forma transparente. Estes componentes incluem serviço de nomes, ORBs (*object request brokers*),

compiladores para linguagem de definição de interface, suporte para localização de objetos, segurança e mecanismos para múltiplos fluxos de execução (*multi-threading*).

Para prover tolerância a falhas de ambiente, dividimos o experimento em duas fases distintas: a primeira fase fornece os serviços da aplicação e a segunda fase estende o modelo gerado pela primeira fase, incorporando tolerância a falhas de ambiente. Em outras palavras, os estados do comportamento (normal e anormal) dos objetos são identificados e uma hierarquia de classes que captura esses diferentes estados é construída em paralelo com a hierarquia de classes da aplicação propriamente dita. As duas hierarquias são ligadas por um mecanismo de delegação.

Podemos concluir de nossa experiência que a utilização de hierarquia de estados e delegação facilita não apenas a provisão de tolerância a falhas de ambiente, mas também a adição de novos requisitos como, por exemplo, distribuição. Com estes mecanismos, mudanças e extensões são realizadas de maneira incremental e, portanto, alterações são introduzidas gradualmente, permitindo um melhor controle da complexidade do sistema.

Capítulo 5

Estruturando um *Framework* para o subdomínio de Controladores de Trens

O estudo de caso apresentado no Capítulo 4 foi projetado como base para avaliar os métodos e técnicas de desenvolvimento de software orientado a objetos. O modelo desenvolvido demonstrou-se bastante flexível e de fácil extensão. As mudanças e extensões para atender os requisitos de tolerância a falhas e distribuição foram realizadas de uma maneira organizada e incremental, especialmente devido ao uso do mecanismo de delegação. A abordagem utilizada permitiu também o desenvolvimento de componentes de software facilmente reutilizáveis, até mesmo em outros domínios de aplicação. Entretanto, como mencionamos no Capítulo 2 (Seção 2.4), tem sido observado que um alto grau de reutilização de software só é alcançado através da reutilização de conjuntos de classes inter-relacionadas e, em muitos casos, de arquiteturas inteiras para um domínio específico, ao invés de reutilizar apenas componentes de software isolados. *Frameworks* orientados a objetos fornecem a funcionalidade necessária para a obtenção desta reutilização de software em grande escala.

Um *framework* provê uma funcionalidade, onde certos pontos são fixos e não podem ser mudados e outros pontos são, entretanto, adaptáveis e destinam-se a acomodar mudanças e extensões. Diferentes programas podem ser criados a partir de um *framework*, dependendo de como os pontos adaptáveis são preenchidos. Portanto, dizemos que um *framework* é utilizado para criar uma família de programas dentro de um determinado domínio [Sch96a]. A terminologia definida por Pree [Pre95], descrita no Capítulo 2 (Seção

2.6), denomina os pontos adaptáveis de um *framework* de "*hot spots*" e os pontos fixos de "*frozen spots*".

Os pontos fixos de um *framework* determinam a arquitetura das aplicações programadas a partir dele. Eles definem a estrutura geral, sua divisão em classes e objetos, as responsabilidades e colaborações entre estas classes e objetos, bem como o fluxo de controle. Os pontos fixos predefinem esses parâmetros de projeto de forma que o projetista ou programador possa concentrar-se nos aspectos específicos de sua aplicação, que são definidos completando-se os pontos adaptáveis. Desta forma, um *framework* proporciona a reutilização não só de código mas também de projeto e, portanto, consiste numa promissora técnica de reutilização de software em grande escala.

Frameworks são classificados, de acordo com o seu conteúdo, em *frameworks* de aplicação e *frameworks* de domínio específico. Um *framework* de aplicação encapsula a funcionalidade básica aplicável a uma grande variedade de aplicações em diferentes domínios como, por exemplo, os *frameworks* para desenvolvimento de interface gráfica. Um *framework* de domínio específico encapsula a funcionalidade de um problema particular de um determinado domínio como, por exemplo, um *framework* para controle de manufatura [Tal94].

Quanto à maneira pela qual uma aplicação é criada, os *frameworks* são classificados em *frameworks* caixa-branca e *frameworks* caixa-preta [JF88]. Um *framework* caixa-branca provê classes que são incompletas com relação aos pontos adaptáveis. Uma aplicação é desenvolvida de um *framework* caixa-branca, derivando classes específicas de classes abstratas e completando ou redefinindo seus métodos. Uma desvantagem neste caso é que torna-se necessário muito conhecimento da estrutura (inclusive implementação) do *framework*. Por outro lado, um *framework* caixa-preta contém todo o código, ou seja, contém para cada ponto adaptável um conjunto de classes alternativas. Uma aplicação é desenvolvida selecionando-se para cada ocorrência de um ponto adaptável, uma ou mais classes, possivelmente parametrizando e configurando-as. Neste caso, o usuário do *framework*, que compõe uma aplicação a partir dele, necessita de pouco conhecimento de engenharia de software, mas deve ter um bom conhecimento do domínio da aplicação.

Não existe necessariamente um limite rígido no uso de *frameworks* caixa-preta e *frameworks* caixa-branca — o espectro entre eles pode ser contínuo, isto é, para o

desenvolvimento de um *framework* específico, dependendo da situação, podemos utilizar as duas técnicas para a construção do mesmo. Quando um *framework* caixa-branca é bem estruturado, alternativas freqüentemente selecionadas podem ser previstas com antecedência e criadas como subclasses completas do tipo caixa-preta, ao passo que os casos menos freqüentes ou não antecipados podem ser criados do tipo caixa-branca, através do desenvolvimento de novas subclasses [Sch96a].

Este capítulo apresenta o processo de desenvolvimento de um *framework* caixa-branca (que inclui também componentes caixa-preta) para o subdomínio específico de Controladores de Trens. O *framework* é estruturado através de uma seqüência de transformações no modelo do Sistema Controlador de Trens (Capítulo 4), utilizando os conceitos de padrões e metapadrões (Seções 2.5 e 2.6). A partir do modelo do Sistema Controlador de Trens, que não provê um grau de reutilização suficientemente alto para um *framework*, algumas transformações são aplicadas, com o objetivo de aumentar o seu grau de reutilização.

O capítulo é organizado da seguinte maneira: Seção 5.1 descreve duas abordagens para o desenvolvimento de *frameworks* baseado em pontos adaptáveis; Seção 5.2 apresenta a descrição do *framework* para o subdomínio de Controladores de Trens; Seção 5.3 apresenta a estrutura geral das principais classes do *framework*; e, por fim, na Seção 5.4 apresentamos algumas conclusões.

5.1 Abordagens Baseadas em Pontos Adaptáveis

O desenvolvimento bem sucedido de um *framework* requer a identificação dos pontos adaptáveis específicos do domínio. Esta tarefa não é trivial, pois requer do projetista do *framework* um grande conhecimento do domínio. Os vários aspectos de um *framework* que não podem ser antecipados para todas as adaptações têm que ser implementados de forma genérica.

Um *framework* bem projetado também predefine a maior parte da arquitetura geral, ou seja, da composição e interação dos componentes. Desta forma, não apenas o código fonte mas todo o projeto da arquitetura — que pode ser considerado a característica mais

importante dos *frameworks* — é reutilizado nas aplicações construídas a parte do *framework*.

Na literatura encontramos duas abordagens principais para o desenvolvimento de *frameworks* baseado em pontos adaptáveis: a abordagem proposta por Pree [Pre95] e a abordagem proposta por Schmid [Sch96a].

Abordagem proposta por Pree

A abordagem baseada em pontos adaptáveis, proposta por Pree [Pre95], pode ser representada como mostra a Figura 50.

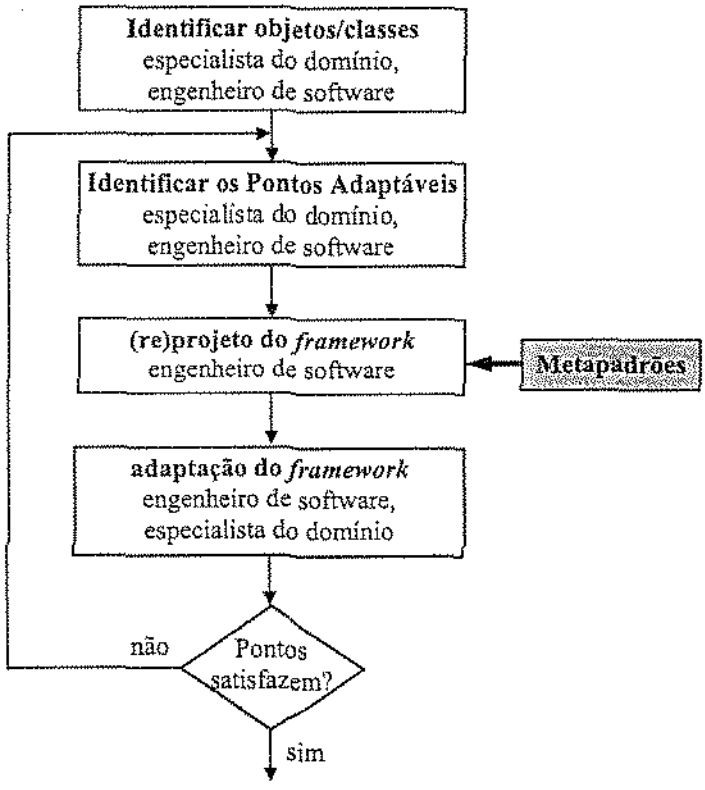


Figura 50 - Abordagem baseada em Pontos Adaptáveis

Uma vez que os pontos adaptáveis desejados são identificados, as características de metapadrões auxiliam no suporte do nível apropriado de flexibilidade. Metapadrões capturam e classificam o projeto de *frameworks*, dando suporte à adaptação destes e ao desenvolvimento de novos *frameworks*.

Abordagem proposta por Schmid

Uma outra abordagem baseada em pontos adaptáveis foi proposta por Schmid [Sch96a], que considera mais útil classificar e descrever a estrutura dos pontos adaptáveis de acordo com aspectos semânticos independentes do domínio, ou seja, através do uso de padrões de projeto. Segundo Schmid, um esforço considerável de desenvolvimento pode ser poupado quando um projetista utiliza padrões para projetar e detalhar a estrutura de pontos adaptáveis do *framework*. O ponto de partida desta abordagem é identificar os pontos adaptáveis e descrever, para cada um deles, o tipo de adaptabilidade requerida. Em seguida, um padrão que provê este tipo de adaptabilidade é selecionado para refinar o ponto adaptável.

Comparação entre as duas abordagens

Comparando as duas abordagens descritas, podemos destacar alguns aspectos importantes:

- Os metapadrões, além de serem aplicados para o projeto dos aspectos independentes de domínio, podem contribuir para documentar o projeto de qualquer *framework* de domínio específico [Pre95];
- Padrões provêem um guia mais concreto para compreensão dos pontos adaptáveis de um *framework* [Sch96a];
- Metapadrões são mais flexíveis. Porém esta vantagem é também uma desvantagem, uma vez que para problemas similares, soluções diferentes podem ser desenvolvidas [Sch96a];
- O catálogo de padrões [GHJV94], como um complemento para as metodologias de análise e projeto orientado a objetos (como é visto por muitos autores), é insuficiente para dar suporte ao desenvolvimento de *frameworks* [Pre95];
- O catálogo de padrões tem-se tornado cada vez mais completo [Sch96a];

Combinação das duas abordagens

A nossa experiência mostra que padrões são realmente mais concretos e, portanto, mais fáceis de manusear e entender. Por outro lado, o catálogo de padrões existente ainda

não é suficiente para documentar todas as adaptabilidades de um *framework* de domínio específico. Desta forma, optamos por estruturar o *framework* para Controladores de Trens, utilizando uma combinação das duas abordagens. Sempre que possível, utilizamos primeiramente padrões, quando estes cobrem a adaptabilidade requerida para um ponto adaptável; caso contrário, utilizamos diretamente os metapadrões. Visto que os padrões podem ser comunicados através de metapadrões, nós procuramos documentar inclusive os padrões através de metapadrões, para obter maior homogeneidade na documentação final do *framework*.

5.2 Descrição do *Framework*

Um *framework* para Sistemas Controladores de Trens permite criar uma família de aplicações correlatas para o subdomínio de controladores de trens apresentado no Capítulo 4. Conseqüentemente, todas as restrições e suposições descritas na Seção 4.1 para este subdomínio permanecem válidas no desenvolvimento do *framework*. O objetivo principal é estruturar o modelo proposto de forma que ele possa ser reutilizado no desenvolvimento de outras aplicações que apresentem variações em alguns aspectos dentro deste subdomínio.

Como dito anteriormente, um domínio consiste de pontos fixos e pontos adaptáveis. No caso do subdomínio de Controladores de Trens, os pontos fixos descrevem as características comuns de diferentes controladores de trens e os pontos adaptáveis descrevem as partes que devem ser flexíveis, podendo ser ajustadas ou redefinidas em diferentes controladores de trens.

Pontos Fixos do *Framework*

Os pontos fixos do *framework* para Controladores de Trens incluem:

- o controle da malha ferroviária;
- o controle do conjunto de trens ou outros objetos móveis;
- o controle dos sensores ou outros dispositivos, que detectam a posição dos trens na malha ferroviária;
- o controle dos atuadores, que efetuam alterações no ambiente;
- a interface do operador;

- a máquina de estados e transições, que é utilizada quando o comportamento do objeto depende de seu estado e pode mudar em tempo de execução;

Pontos Adaptáveis do *Framework*

Os pontos adaptáveis do *framework* para Controladores de Trens incluem:

- a composição da malha ferroviária. Diferentes aplicações possuem diferentes composições para a malha ferroviária; portanto, o *framework* deve dar suporte para a definição de qualquer formato de malha;
- a visão da malha, que deve refletir a composição da malha ferroviária;
- diferentes tipos de componentes: trens, sensores e atuadores. Diferentes aplicações contêm diferentes tipos de trens, sensores e atuadores, os quais apresentam funções similares;
- diferentes tipos de seções. O comportamento dos tipos de seções pode variar entre diferentes aplicações;
- tolerância a falhas. Flexibilidade para a definição dos estados (normais e anormais) dos componentes tolerantes a falhas. A implementação da máquina de estados que executa essas transições é independente e, portanto, faz parte da camada fixa do *framework*;
- Tratamento de eventos. Aplicações específicas podem definir ou redefinir diferentes tipos de tratadores de eventos e/ou eventos;
- Protocolo de comunicação. Diferentes aplicações podem redefinir ou mesmo trocar o protocolo que implementa a camada de comunicação entre objetos distribuídos;

Formato

O formato utilizado nas próximas seções para descrever os pontos adaptáveis consiste de quatro divisões (com base no formato apresentado em [Sch96a]). Estas divisões descrevem:

(i) o ponto adaptável;

(ii) os problemas relacionados a esta adaptabilidade;

(iii) os requisitos a serem cumpridos pela solução;

(iv) o padrão/metapadrão que cobre a adaptabilidade, bem como uma representação gráfica (em notação UML) de como este padrão/metapadrão implementa a modificação na estrutura existente;

5.2.1 Ponto Adaptável para a Composição da Malha

Adaptabilidade

Diferentes aplicações possuem diferentes composições para a malha ferroviária.

Problema

O Controlador de Trens possui uma malha ferroviária composta de três partes, sendo que cada parte é composta basicamente por seções. Cada seção é composta por estações (sensores) e conectores (atuadores). Além disso, temos o conceito de região de controle — conjunto de seções predefinidas para o trajeto do trem — utilizado para dar suporte aos mecanismos de tolerância a falhas. Toda esta composição atual da malha ferroviária corresponde a uma configuração específica.

Requisito

Permitir uma composição flexível da malha ferroviária, incluindo uma clara distinção das regiões de controle e das partes que são gerenciadas por controles distribuídos.

Padrão/metapadrão

O **padrão Composição** [GHJV94, Página 163], como mostra a Figura 51, permite representar a composição de objetos, de forma que os objetos primitivos e compostos possam ser tratados uniformemente.

A classe Item da Malha define a interface para os objetos que compõem a malha ferroviária. A classe Bloco define o comportamento para os objetos compostos: as partes da

malha que terão controle distribuído; as regiões de controle; e, possivelmente, os trajetos ou rotas. Além disso, podemos definir operações a serem executadas uniformemente em objetos primitivos e objetos compostos. Por exemplo, a operação Livre (Figura 51) retorna verdadeiro se o objeto não estiver sendo acessado e falso, caso contrário.

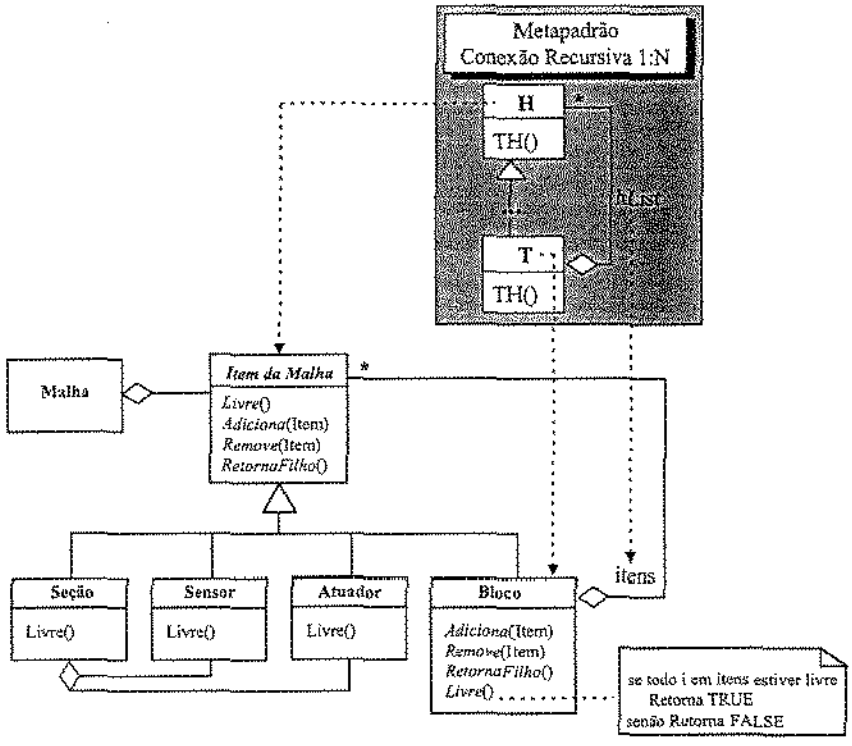


Figura 51 - Estrutura para Composição da Malha Ferroviária

O **metapadrão Conexão Recursiva 1:N** é aplicado à estrutura para composição da malha ferroviária do *framework*. A Figura 51 ilustra como os componentes deste metapadrão correspondem às classes do *framework*.

5.2.2 Ponto Adaptável para a Visão da Malha

Adaptabilidade

O Controlador de Trens é uma aplicação distribuída, ou seja, possui controles distribuídos para diferentes partes da malha ferroviária. A Visão da malha deve refletir o comportamento das diversas partes da malha de maneira flexível e, sobretudo, consistente.

Problema

Quando as partes da malha ferroviária são controladas por controladores distribuídos, várias visões da malha ferroviária poderão estar disponíveis nos diversos nodos da rede. Porém, cada controlador atualiza somente a sua respectiva visão, ou seja, a visão da parte da malha que está sob seu controle.

Requisito

Permitir a atualização das diversas visões disponíveis para a malha ferroviária, independentemente do funcionamento do controlador.

Padrão/metapadrão

O **metapadrão Unificação Recursiva 1:N** é aplicado à estrutura de atualização das visões da malha ferroviária. A Figura 52 mostra como os componentes deste metapadrão correspondem às classes do *framework*.

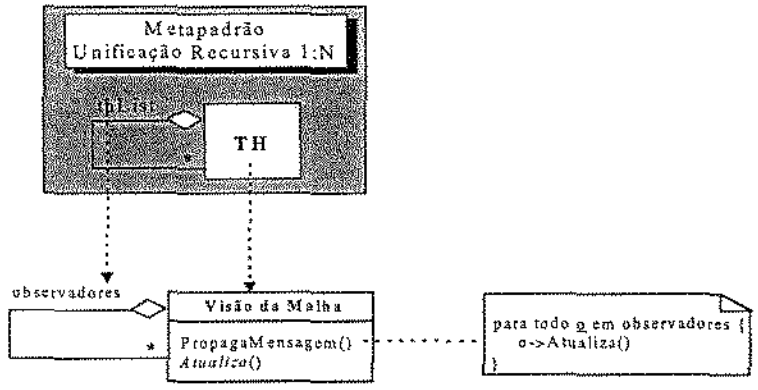


Figura 52 - Estrutura para Atualização das Visões da Malha Ferroviária

Os aspectos essenciais da aplicação do metapadrão Unificação Recursiva 1:N para a estrutura de atualização das visões da malha ferroviária podem ser expressos textualmente da seguinte maneira:

Metapadrão: Unificação Recursiva 1:N
TH -> Visão da Malha
T() -> PropagaMensagem()
H() -> Atualiza()

thList -> observadores

ponto adaptável: atualização das visões da malha independente das operações de controle

Esta abordagem é uma variação do **padrão Observador** [GHJV94, Página 293], tendo como observado e observador, um objeto da própria classe Visão da Malha. Os objetos Visão da Malha observando um objeto Visão da Malha particular são controlados pelos métodos:

`void AdicionaVisão(Visão *v)` //adiciona um objeto na estrutura observadores

`Visão *RemoveVisão(Visão *v)` //remove um objeto da estrutura observadores

O fato do metapadrão Unificação Recursiva 1:N ser utilizado implica que uma hierarquia de objetos dependentes pode ser definida: objetos que observam um objeto particular podem, por sua vez, ser observados por outros objetos.

5.2.3 Ponto Adaptável para Criação de Componentes

Adaptabilidade

Os componentes da ferrovia — trens, atuadores e sensores — são objetos de configuração específica. Diferentes aplicações contêm diferentes tipos de trens, atuadores e sensores, os quais apresentam funções similares.

Problema

Os componentes Trem, Estação e Conector são específicos do Controlador de Trens. Uma aplicação cria trens da classe Trem, atuadores da classe Conector e sensores da classe Estação. Porém, outros tipos de trens, atuadores e sensores apresentam funcionalidade diferente e não podem ser criados sem alterar a implementação dos componentes existentes.

Requisito

Permitir a criação flexível dos componentes trem, atuador e sensor. Desta forma, o *framework* torna-se independente de como os objetos são criados.

Padrão/metapadrão

O **padrão Protótipo** [GHJV94, Página 117], como mostra a Figura 53, especifica os tipos de objetos a serem criados utilizando-se um protótipo e cria novos objetos a partir da cópia ("clonagem") deste protótipo. O *framework* fornece uma classe abstrata Objeto Móvel, que representa os trens e outros possíveis objetos móveis da ferrovia como, por exemplo, vagonetes. O *framework* predefine também uma classe Gerente Objeto Móvel, que é responsável por criar instâncias de objetos móveis. Esta classe provê a operação ConstróiObjetoMóvel, que cria um novo objeto da seguinte maneira:

- inicializa um apontador para uma classe concreta (subclasse da classe Objeto Móvel) que implementa a funcionalidade do componente;
- solicita a execução da operação Clone nesta classe concreta (por exemplo, Trem);
- a operação Clone retorna uma cópia ("clone") do próprio objeto.

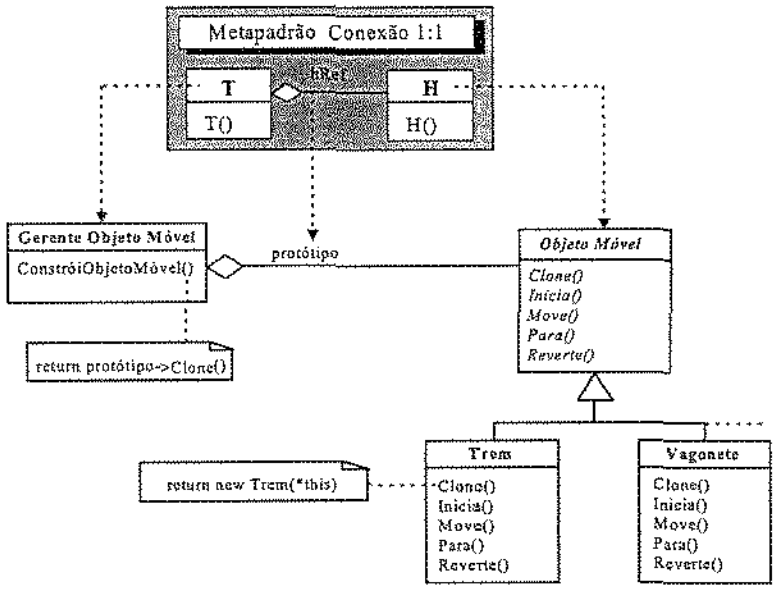


Figura 53 - Criação Flexível de Objetos Móveis

Em síntese, Gerente Objeto Móvel cria um novo Objeto Móvel copiando uma instância de uma subclasse de Objeto Móvel. Nós chamamos esta instância de protótipo. Gerente

Objeto Móvel é parametrizada pelo protótipo que ele utiliza para cópia. Portanto, todas as subclasses de Objeto Móvel devem prover uma operação Clone[6].

O Controle Central utiliza um Gerente Objeto Móvel para inserir um determinado tipo de objeto móvel na malha ferroviária da seguinte maneira:

```
ControleCentral::InsereObjetoMóvel(GerenteObjetoMóvel
*gm) {
    ObjetoMóvel *om;
    om = gm->ConstróiObjetoMóvel(); ...}
```

Nós podemos utilizar Gerente Objeto Móvel para criar um objeto móvel padrão, simplesmente iniciando-o com um protótipo da classe Trem:

```
GerenteObjetoMóvel gmPadrão(new Trem);
controleCentral->InsereObjetoMóvel(gmPadrão);
```

Para mudar o tipo de objeto móvel, nós iniciamos Gerente Objeto Móvel com um protótipo diferente. Por exemplo, a seguinte chamada cria um objeto móvel Vagonete.

```
GerenteObjetoMóvel gmNovo(new Vagonete);
controleCentral->InsereObjetoMóvel(gmNovo);
```

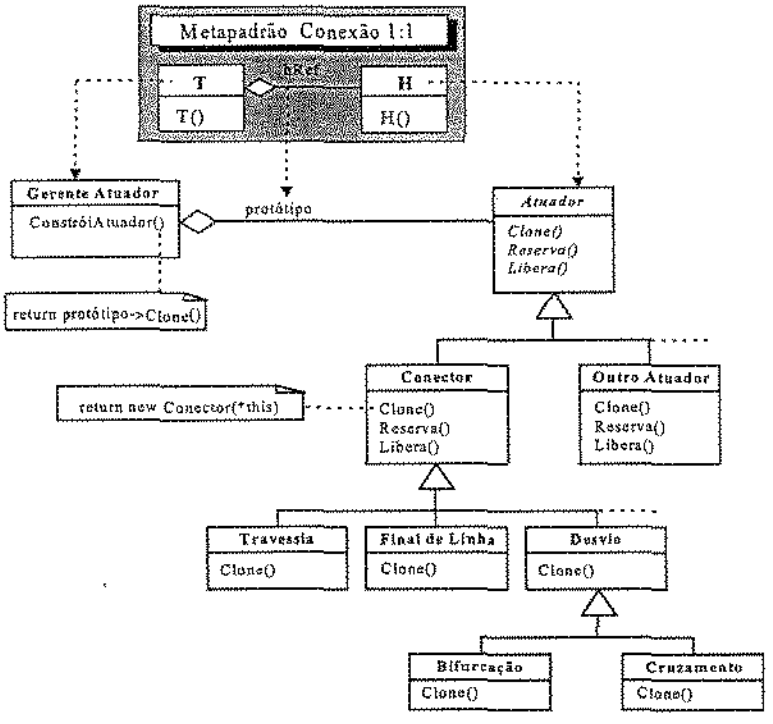


Figura 54 - Criação Flexível de Atuadores

[6] Maiores detalhes sobre aspectos de implementação da operação Clone são discutidos em [GHJV94, Página 121].

O mesmo padrão é aplicado para criação flexível dos componentes Atuador (Figura 54) e Sensor (Figura 55).

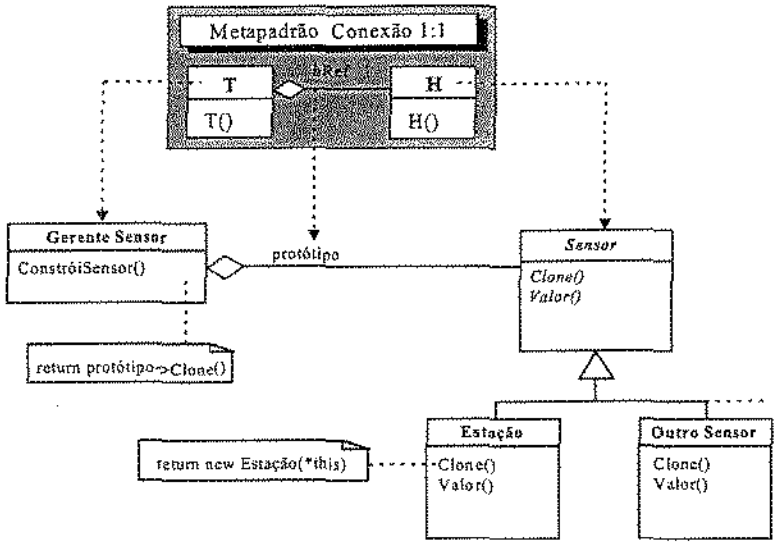


Figura 55 - Criação Flexível de Sensores

O **metapadrão Conexão 1:1** é aplicado às estruturas dos componentes do *framework* (objetos móveis, atuadores e sensores). A Figura 53, a Figura 54 e a Figura 55 ilustram como os componentes deste metapadrão correspondem às classes do *framework*.

5.2.4 Ponto Adaptável para Tipos de Seções

Adaptabilidade

O Controlador de Trens provê três tipos básicos de seções na malha ferroviária: seção sólida, seção particionada e seção interconectada. O comportamento destes tipos de seções pode variar entre diferentes aplicações. Além disso, um objeto Seção pode alterar o seu comportamento em tempo de execução, devido a mudanças no tipo de seção.

Problema

Uma seção precisa mudar de comportamento. Por exemplo, uma seção sólida assume o comportamento de uma seção do tipo particionada, devido a interrupções, tais como, a queda de uma árvore, a presença de animais ou um acidente no meio de uma seção,

bloqueando a passagem de trens. Outro exemplo, a fronteira entre as partes da malha pode ser redefinida e, assim, uma seção sólida pode passar a ser uma seção interconectada.

Estes e outros exemplos de mudança de estado entre seções só podem ser tratados de maneira eficiente se os estados forem controlados em tempo de execução.

Requisito

Tornar flexível o comportamento dos tipos de seções e permitir que este comportamento seja alterado em tempo de execução.

Padrão/metapadrão

O **padrão Estado** [GHJV94, Página 305], como mostra a Figura 56, permite a um objeto alterar seu comportamento quando seu estado interno muda.

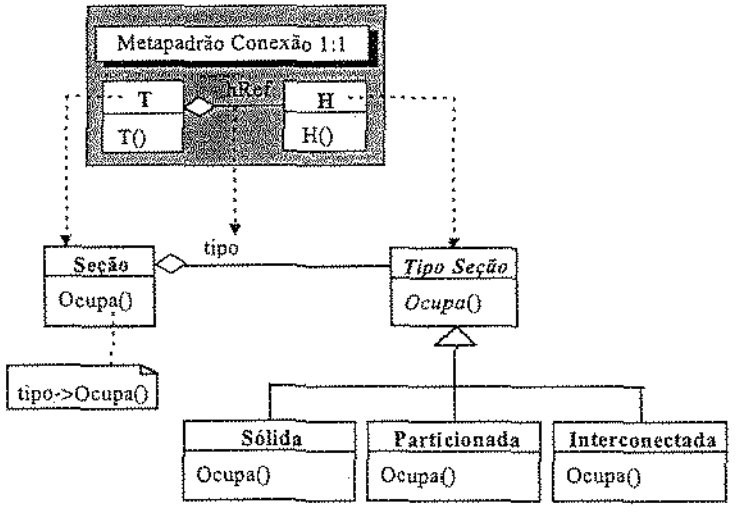


Figura 56 - Estrutura para Tipos de Seções

Um objeto Seção delega mensagens para o objeto Tipo Seção: um objeto "interno" que muda dinamicamente de estado (Seção Sólida, Seção Particionada e Seção Interconectada), de acordo com as mudanças no tipo de seção. Uma operação (por exemplo, Ocupa) é enviada, em tempo de execução, para o objeto que representa o estado corrente (apontador tipo na Figura 56). Nesta solução, o estado lógico de um objeto Seção pode mudar sem a necessidade de excluir e recriar o objeto, ou seja, quando uma seção

muda de tipo, o apontador para o tipo corrente aponta para a instância da subclasse que implementa o novo tipo.

O padrão Estado não especifica qual participante define o critério para as transições de estado. No caso dos tipos de seções, o critério para as transições de estado é definido externamente (através da Interface do Operador). A implementação destas transições deve ser provida na parte fixa do *framework*, com base na materialização dos eventos (veja Seção 5.2.6). Por exemplo, um evento do tipo ObstruirSeção gerado na Interface do Operador causa a transição SeçãoSólida -> SeçãoParticionada. Esta abordagem permite reutilizar a máquina de estados em outras partes do *framework* como, por exemplo, na implementação dos estados e transições para tolerância a falhas de ambiente (veja Seção 5.2.5).

O **metapadrão Conexão 1:1** é aplicado à estrutura para tipos de seções do *framework*. A Figura 56 ilustra como os componentes deste metapadrão correspondem às classes do *framework*.

5.2.5 Ponto Adaptável para Tolerância a Falhas

Adaptabilidade

Para prover tolerância a falhas de ambiente, consideramos duas fases de comportamento para os objetos tolerantes a falhas do Controlador de Trens — fase normal e fase anormal. O usuário do *framework* deve ter flexibilidade para (re)definir a funcionalidade dos objetos, tanto para a fase normal quanto para a fase anormal.

Problema

Sensores e atuadores formam a base para a definição dos tipos de falhas de ambiente presentes no sistema. A estrutura de tolerância a falhas deve ser suficientemente flexível e extensível, permitindo manter a complexidade sob controle, ao passo que os casos anormais ou excepcionais são considerados.

Requisito

Separar o comportamento normal e anormal dos objetos tolerantes a falhas, de forma a separar as atividades normais das anormais relacionadas com mecanismos de tolerância a falhas.

Padrão/metapadrão

O **padrão Estado** [GHJV94, Página 305], como mostra a Figura 57 e a Figura 58, permite criar uma nova hierarquia de estados para modelar claramente os estados normal e anormal de atuadores e sensores. A classe Atuador (Figura 57) delega requisições específicas para o corrente objeto EstadoAtuador, de acordo com o seu estado interno (normal ou anormal). A mesma abordagem é utilizada para a classe Seção (Figura 58).

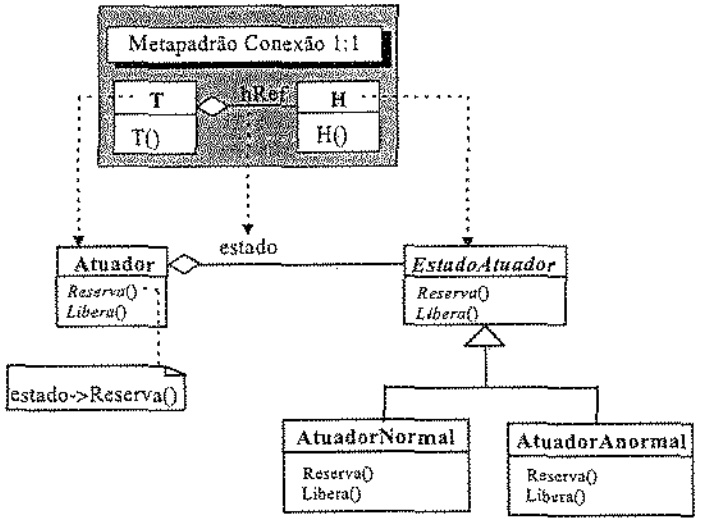


Figura 57 - Estrutura para Tolerância a Falhas em Atuadores

O padrão Estado atribui a um objeto todo o comportamento associado a um estado particular. Devido a todo código específico de um estado estar encapsulado em uma subclasse da classe Estado (por exemplo, EstadoAtuador), novos estados e transições podem ser facilmente adicionados através da definição de novas subclasses.

Esta abordagem para estruturar tolerância a falhas no Controlador de Trens (incluindo um mecanismo sofisticado de tratamento de exceções) foi utilizada intuitivamente no modelo definido no Capítulo 4, com base no trabalho desenvolvido por [Rub94]. Desta

forma, para este aspecto do *framework*, a única transformação realizada foi a associação do padrão correspondente.

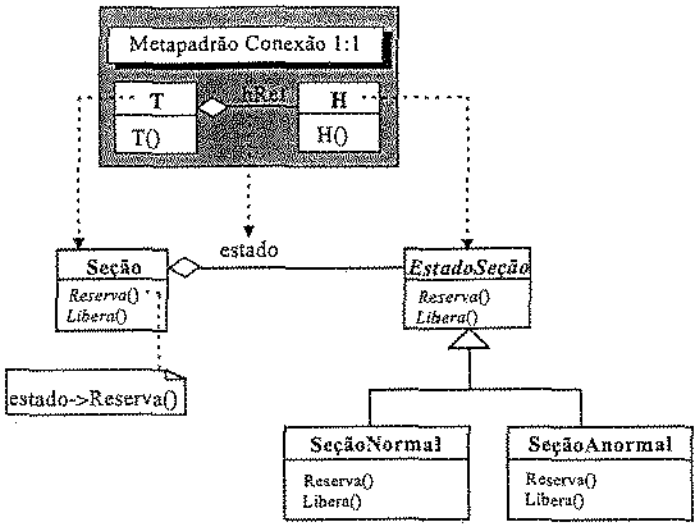


Figura 58 - Estrutura para Tolerância a Falhas em Seções

O **metapadrão Conexão 1:1** é aplicado à estrutura para tolerância a falhas do *framework*. A Figura 57 e a Figura 58 ilustram como os componentes deste metapadrão correspondem às classes do *framework*.

5.2.6 Ponto Adaptável para Tratamento de Eventos

Adaptabilidade

O Controlador de Trens é um sistema orientado a eventos. Desta forma, aplicações específicas podem definir (ou redefinir) diferentes tipos de tratadores de eventos e/ou eventos, com o objetivo de atender requisitos específicos de cada aplicação.

Problema

Toda funcionalidade do modelo definido para o Sistema Controlador de Trens concentra-se na interface da classe Controle Central. Esta classe, além de tratar seus próprios eventos (por exemplo, iniciar/parar o sistema e inserir/remover trens), funciona como intermediária para o tratamento dos principais eventos do sistema. Isto impossibilita a

uma aplicação específica definir novos tipos de tratadores e/ou eventos, sem alterar a interface da classe Controle Central.

Requisito

Adicionar flexibilidade na estrutura de tratamento de eventos do *framework* — materializar os eventos — e, com isso, simplificar a interconexão entre os objetos.

Padrão/metapadrão

O **padrão Reator** [Sdt95a, Sdt95b], como mostrado na Figura 59, registra e ativa múltiplos tratadores de eventos. Este padrão provê várias vantagens para aplicações orientadas a eventos. Ele facilita o desenvolvimento de aplicações flexíveis, pois permite ao *framework* encapsular a captação de eventos, bem como a ativação dos tratadores apropriados para tais eventos. Desta forma, possibilita à aplicação acrescentar funcionalidade específica, através da (re)definição dos métodos dos tratadores de eventos. Além disso, o padrão Reator facilita a extensibilidade da aplicação, uma vez que os tratadores de eventos podem ser desenvolvidos independentemente dos mecanismos de captação de eventos.

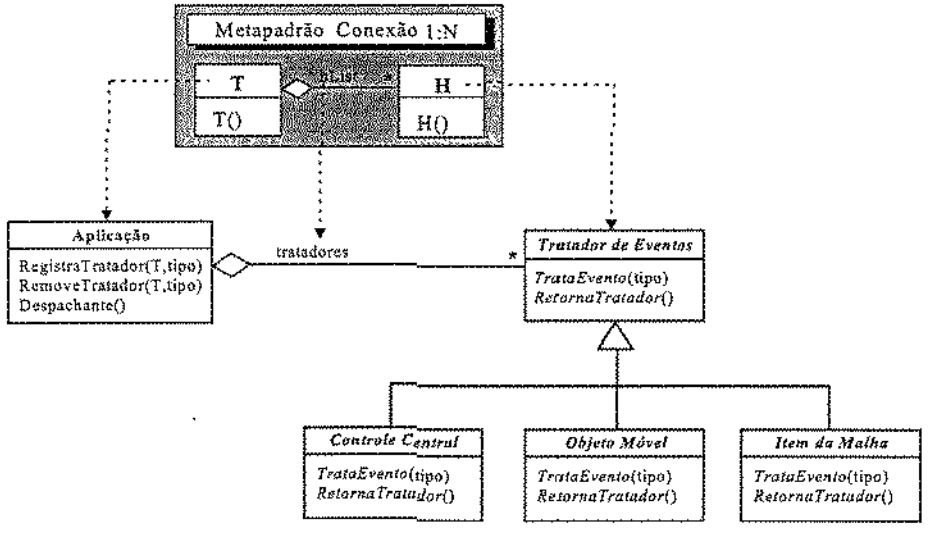


Figura 59 - Estrutura para Tratamento de Eventos

A classe Aplicação define uma interface para registrar, remover e despachar objetos da classe Tratador de Eventos. O método Despachante provê o seguinte laço para captação de eventos[7]:

```
seleciona(tratadores)
para cada t em tratadores {
    t->TrataEvento(tipo)
}
```

A classe Tratador de Eventos especifica uma interface usada pela Aplicação para ativar métodos definidos por objetos que são pré-registrados para tratar determinados eventos. As subclasses de Tratador de Eventos implementam os métodos que realizam o processamento propriamente dito dos eventos.

Essas classes colaboram, em termos gerais, da seguinte maneira:

- A Aplicação ativa o método TrataEvento dos objetos da classe Tratador de Eventos, em resposta à captação de eventos. Estes eventos são associados à fonte de evento Interface do Operador. Para ligar a Aplicação aos tratadores de eventos, as subclasses de Tratador de Eventos devem sobrepor o método RetornaTratador. Quando a Aplicação registra um objeto de uma determinada subclasse da classe Tratador de Eventos, ela obtém o apontador para o objeto invocando o método Tratador de Eventos-> RetornaTratador;
- Quando um evento ocorre, a Aplicação usa o apontador ativado pelo evento como chave para localizar e ativar o método apropriado no Tratador de Eventos. O método TrataEvento é chamado pela Aplicação para realizar a funcionalidade específica em resposta a um determinado evento;

A interface da classe base Tratador de Eventos contém apenas um método TrataEvento usado pela Aplicação para tratar eventos. Neste caso, o tipo do evento é passado como parâmetro para o método[8]. Esta abordagem torna possível adicionar novos tipos de eventos sem modificar a interface. Entretanto, esta abordagem provoca o uso de expressões condicionais nos métodos das subclasses da classe Tratador de Eventos, o que

[7] Maiores detalhes sobre aspectos de implementação da captação de eventos são discutidos em [Sdt95a] e [GHJV94, Página 226].

[8] A passagem do tipo de evento como parâmetro para o método TrataEvento implica no empacotamento e desempacotamento dos argumentos do evento. Maiores detalhes sobre estes aspectos de implementação são discutidos em [GHJV94, Página 226].

de certa forma, dificulta a extensibilidade. O método TrataEvento da subclasse Controle Central, por exemplo, seria definido da seguinte maneira:

```
void ControleCentral::TrataEvento(tipo) {
    switch(tipo) {
    case iniciaSistema        : //...
    case finalizaSistema      : //...
    case insereObjetoMóvel    : //...
    case removeObjetoMóvel    : // ...
    default                   : //...
    }
}
```

Uma outra alternativa para implementar a interface da classe Tratador de Eventos é definir métodos separados para cada tipo de evento. Esta abordagem não envolve expressões condicionais, porém requer ao projetista do *framework* prever antecipadamente o conjunto completo de métodos da classe Tratador de Eventos. Esta abordagem não resolveria o nosso problema, uma vez que desejamos flexibilidade para a usuário do *framework*, na definição dos eventos.

O **metapadrão Conexão 1:N** é aplicado à estrutura de tratamento de eventos do *framework*. A Figura 59 ilustra como os componentes deste metapadrão correspondem às classes do *framework*.

5.2.7 Ponto Adaptável para o Protocolo de Comunicação

Adaptabilidade

O usuário do *framework* deve ter flexibilidade para redefinir ou mesmo trocar o protocolo de comunicação.

Problema

O Controlador de Trens é uma aplicação distribuída e, portanto, precisa prover uma série de requisitos para manter objetos distribuídos através de uma rede de computadores.

Requisito

Fornecer uma interface transparente para a comunicação entre os objetos distribuídos das aplicações.

Padrão/metapadrão

O **padrão** *Proxy* **Remoto** [GHJV94, Página 207], como mostra a Figura 60, provê uma representação local para um objeto distribuído. Este *padrão* encapsula o fato de um objeto pertencer a outro espaço de endereçamento. Todo evento a ser tratado por um objeto da classe ProtocoloComunicação é empacotado junto com seus argumentos e enviado para o objeto real (uma subclasse da classe Tratador de Eventos) em outro espaço de endereçamento.

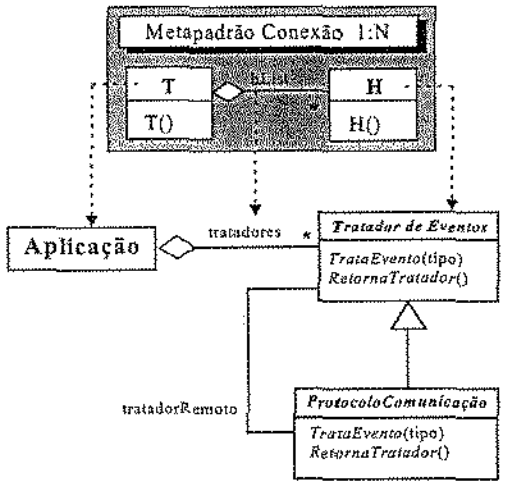


Figura 60 - Estrutura para o Protocolo de Comunicação

Na estrutura básica do padrão *Proxy*, o objeto intermediário (*proxy*) e o objeto real são subclasses de uma mesma classe C, a qual define uma interface comum para estes objetos, de forma que o *proxy* possa ser utilizado em qualquer lugar onde um objeto real é esperado. Entretanto, o *proxy* ProtocoloComunicação não precisa conhecer o tipo do objeto real. Portanto, o ProtocoloComunicação mantém uma referência para um objeto da classe abstrata Tratador de Eventos, que é ligado ao objeto real em tempo de execução. Por exemplo, quando um evento InsereObjetoMóvel é requerido a um objeto ControleCentral (subclasse da classe Tratador de Eventos) que se encontra em outro espaço de

endereçamento, o evento é primeiramente tratado pelo objeto ProtocoloComunicação, que o envia para ser tratado pelo objeto real ControleCentral.

O **metapadrão Conexão 1:N** é aplicado à estrutura para o protocolo de comunicação do *framework*. A Figura 60 ilustra como os componentes deste metapadrão correspondem às classes do *framework*.

Quanto à criação do Protocolo de Comunicação, podemos aplicar o **metapadrão Unificação** para obter uma criação flexível deste objeto (Figura 61). A classe Minha Aplicação sobrepõe o método CriaProtocolo para retornar uma instância da classe ProtocoloRPC. Desta forma, podemos adaptar o tipo de protocolo de comunicação conforme os requisitos da aplicação específica (Minha Aplicação). O *framework* fornece o protocolo baseado no mecanismo de RPC como padrão, mas protocolos baseados em outros mecanismos podem ser desenvolvidos e adaptados nas aplicações construídas a partir do *framework*.

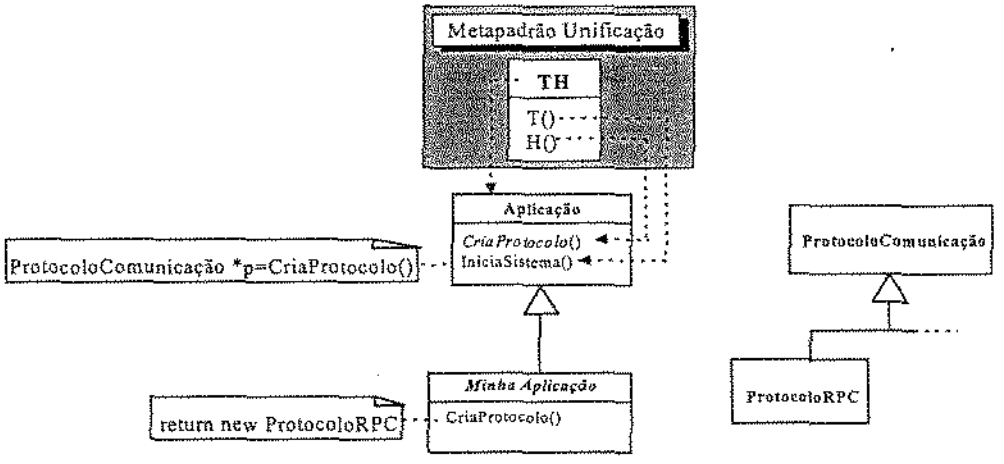


Figura 61 - Criação Flexível do Protocolo de Comunicação

5.3 Estrutura Geral do *Framework*

A Figura 62 apresenta a estrutura geral das principais classes do *framework*. De maneira semelhante à abordagem encontrada no *framework* ET++ [WG94], todas as principais classes do *framework* para Controladores de Trens são derivadas de uma classe base Objeto e, portanto, compartilham seu comportamento. A classe Objeto define e implementa

parcialmente alguns serviços, em especial, a infra-estrutura para solicitação de meta-informação (veja Seção 5.3.1).

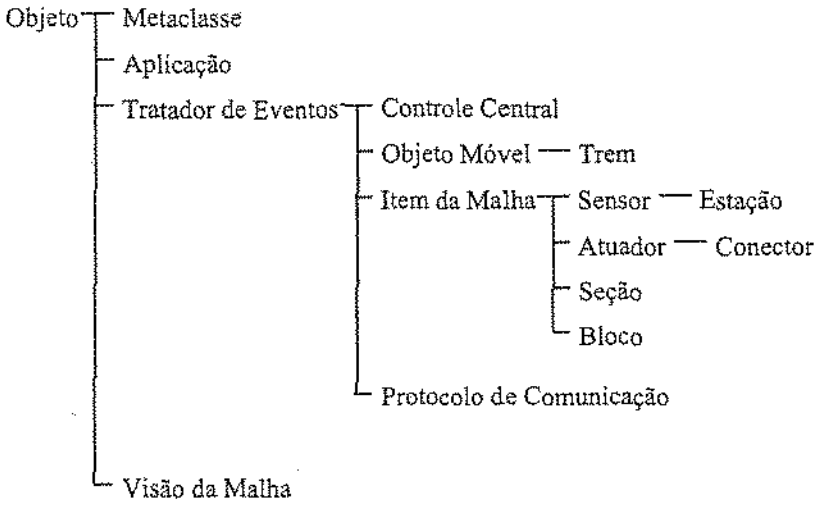


Figura 62 - Estrutura Geral do *Framework*

Toda aplicação construída a partir do *framework* tem exatamente um objeto da classe Aplicação (Figura 62). O passo inicial da execução de uma aplicação consiste em criar um objeto Aplicação e invocar o método Executa neste objeto. Desta forma, inicia-se o fluxo de eventos.

5.3.1 Meta-Informação

Em geral, as linguagens de programação orientadas a objetos, como C++, não provêem meta-informações, isto é, informações sobre a classe ou atributos de um determinado objeto. Porém, estas informações são muito úteis para o sistema de suporte a execução do *framework*.

Uma abordagem para prover suporte a meta-informações, extraída do *framework* ET++ [WG94], consiste em associar, a toda classe do *framework*, um objeto especial contendo meta-informação sobre a classe. Este objeto é uma instância da classe MetaClasse (Figura 62) e armazena informações, tais como, o nome da classe, o nome da sua superclasse, o tamanho de uma instância da classe em *bytes* e os nomes e tipos de seus atributos. Esta informação pode ser acessada através de métodos oferecidos na classe

Objeto (Figura 62). Desta forma, meta-informações podem ser recuperadas para todas as instâncias de classes descendentes da classe Objeto.

Alguns métodos oferecidos pela classe Objeto para recuperar meta-informação são:

- ÉdoTipo(NomeClasse): retorna VERDADEIRO se o objeto cujo método ÉdoTipo(...) é chamado for uma instância da classe NomeClasse ou uma instância de uma classe descendente de NomeClasse;
- MetaObjeto(): retorna um apontador para a instância da classe MetaClasse que contém a meta-informação do objeto;
- NomeClasse(): retorna o nome da classe do objeto;

Por exemplo, supondo uma declaração do tipo:

```
Seção *s = new Seção();
```

A chamada s->MetaObjeto()->Tamanho() retorna o tamanho em *bytes* de uma instância da classe Seção. A operação Tamanho é provida na classe MetaClasse.

5.4 Conclusões

Nós mostramos como um *framework* para o subdomínio de Controladores de Trens pode ser construído através da utilização de padrões de projeto e metapadrões. Mais especificamente, o *framework* foi obtido através de uma seqüência de transformações do modelo do Sistema Controlador de Trens (Capítulo 4), utilizando-se os conceitos de padrões e metapadrões. A partir do modelo do Sistema Controlador de Trens, que não proporcionou um grau de reutilização suficientemente alto para um *framework*, algumas transformações foram aplicadas, com o objetivo de aumentar o seu grau de reutilização.

A nossa experiência verificou que a grande dificuldade na construção de um *framework* consiste em determinar quais os pontos que devem ser deixados flexíveis e quais os pontos devem ser fixos. Para obter um "balanceamento" correto entre os pontos fixos e os pontos adaptáveis requer-se um profundo conhecimento do domínio específico da aplicação e também numerosas iterações do projeto do *framework*. Verificamos também que padrões de projeto e metapadrões são técnicas efetivas para se estruturar os pontos adaptáveis de um *framework*. Estas técnicas são apropriadas para desenvolver e documentar

o projeto de um *framework* de maneira eficiente e num nível de abstração independente dos detalhes de linguagens de programação.

Capítulo 6

Conclusões

Esta dissertação concentrou-se na estruturação de aplicações distribuídas e tolerantes a falhas de ambiente utilizando técnicas orientadas a objetos, tais como: abstração de dados, compartilhamento de comportamento (incluindo herança e delegação), classes abstratas, polimorfismo e acoplamento dinâmico. Para o entendimento e validação dessas técnicas foi desenvolvido um protótipo de uma aplicação distribuída orientada a objetos: um Controlador de Trens.

O requisito de distribuição foi adicionado de uma maneira incremental ao modelo. Orientação a objetos provê computação distribuída com os mesmos benefícios que são providos para computação não distribuída: encapsulamento, reutilização, portabilidade, extensibilidade, etc. A implementação de um protocolo para prover a comunicação entre os objetos distribuídos mostrou-se uma solução apropriada para encapsular os mecanismos de distribuição. Esta solução foi definida de maneira ortogonal e, portanto, facilita mudanças na implementação dos mecanismos de distribuição sem afetar o modelo.

Para prover tolerância a falhas de ambiente, dividimos o experimento em duas fases distintas: a primeira fase fornece os serviços da aplicação e a segunda fase estende o modelo gerado pela primeira fase, incorporando tolerância a falhas de ambiente. Em outras palavras, os estados do comportamento (normal e anormal) dos objetos são identificados e uma hierarquia de classes que captura esses diferentes estados é construída em paralelo com a hierarquia de classes da aplicação propriamente dita. As duas hierarquias são ligadas por um mecanismo de delegação.

Por fim, nós mostramos como um *framework* orientado a objetos para o subdomínio de Controladores de Trens pode ser construído através da utilização de padrões de projeto e metapadrões. A nossa experiência verificou que a grande dificuldade na construção de um *framework* consiste em determinar quais os pontos que devem ser deixados flexíveis e quais os pontos devem ser fixos. De fato, obter um "balanceamento" correto entre os pontos fixos e os pontos adaptáveis requer um profundo conhecimento do domínio específico da aplicação e também numerosas iterações do projeto do *framework*.

A seguir discutimos alguns trabalhos relacionados. Em seguida, sugerimos alguns trabalhos que julgamos serem relevantes como objetos para pesquisas futuras.

6.1 Trabalhos Relacionados

Esta seção discute alguns trabalhos da literatura relacionados. O primeiro, denominado "máquinas de estados orientadas a objetos", está relacionado com a utilização de hierarquia de estados e delegação. Os demais estão diretamente relacionados com a estruturação de *frameworks*.

Máquinas de Estados Orientadas a Objeto

Sane e Campbell [SC95] propõem uma técnica orientada a objetos para especificação e implementação de software com base em máquinas de estados. Esta técnica permite a derivação de máquinas de estados complexas a partir de máquinas de estados simples. Para tanto, são utilizados os conceitos de subclasses, composição, delegação e classes genéricas, os quais modificam as máquinas de estados simples de maneira incremental.

Como vimos, o nosso trabalho propôs a criação de hierarquias de estados e delegação para prover tolerância a falhas de ambiente. Embora as hierarquias de estados do modelo desenvolvido tenham sido implementadas através de máquinas de estados simplificadas, estas máquinas de estados podem aumentar bastante a sua complexidade mediante novos requisitos. Neste caso, poderíamos utilizar a abordagem proposta por Sane e Campbell para tratar problemas como, por exemplo, a representação das transições de estados nas classes derivadas.

Framework para Sistemas de Controle de Navios

Dagermo e Knutsson [DK96] descrevem a experiência de desenvolvimento de um *framework* para sistemas de controle de navios utilizando uma série de padrões de projetos. Este *framework* inclui um conjunto de objetos que são usados para construir sistemas de controle de navios, um modelo de execução, mecanismos de distribuição e suporte para arquivos de configuração.

Um sistema de controle desenvolvido a partir do *framework* deve ser projetado com a mínima consideração possível em relação à distribuição física das funções do sistema. Para tanto, o *framework* inclui um mecanismo de rede que torna a distribuição física dos objetos transparente, deixando as decisões referentes à distribuição para o último passo do projeto. O padrão *Proxy* [GHJV94, Página 207] é utilizado no projeto para tornar transparente o acesso a objetos localizados em diferentes nodos.

Segundo Dagermo e Knutsson, a introdução de padrões de projetos na engenharia de software, e particularmente em projeto orientado a objetos, representa um grande passo em direção à produção de software reutilizável e de alta qualidade.

Framework para Controle de Manufatura

Schmid [Sch95, Sch96a, Sch96b] apresenta o projeto de um *framework* caixa-preta para sistemas de controle automatizado de manufatura utilizando padrões de projeto. Neste *framework*, uma aplicação é criada para uma configuração específica através de um processo de construção: os componentes reutilizáveis são selecionados do *framework* caixa-preta e configurados de acordo com a aplicação específica.

Uma provável evolução do *framework* proposto no capítulo 5 seria a definição de todos os componentes como caixa-preta. Mas é importante ressaltar que o custo de desenvolvimento de um *framework* caixa-preta é muito maior do que o custo de desenvolvimento de um *framework* caixa-branca, uma vez que todos os pontos adaptáveis têm que ser identificados e providos a priori. Portanto, a nossa proposta inicial de implementar como caixa-preta apenas alguns componentes padrões demonstra-se bastante apropriada.

Segundo Schmid, padrões de projeto provêem grande ajuda para gerenciar a complexidade das soluções orientadas a objetos, tanto como um guia durante o processo de desenvolvimento quanto para um melhor entendimento do projeto final.

Aplicação de Metapadrões no *Framework* ET++

Pree [Pre95, Capítulo 5] utiliza o *framework* ET++ para demonstrar as vantagens de se aplicar metapadrões em um *framework* complexo. ET++ [WG94] é um *framework* caixa-branca para desenvolver aplicações de interface gráfica com usuário (em inglês, *GUI - Graphic User Interface*). Ele provê desde componentes elementares para construção de interfaces, tais como, botões e menus, até componentes de aplicação de alto nível como janelas e documentos.

Pree utiliza metapadrões para descrever o projeto de alguns dos principais mecanismos de ET++. Segundo ele, metapadrões representam uma excelente maneira de estruturar o projeto de um *framework* de maneira eficiente e num nível de abstração bem mais alto do que uma linguagem de programação. Além disso, eles podem ser aplicados em qualquer *framework*.

6.2 Trabalhos Futuros

Alguns tópicos importantes relacionados a esta dissertação que nós julgamos serem relevantes como objetos de trabalhos futuros são: implementação do *framework*, uso de reflexão computacional, implementação de algoritmos de caminhamento na malha ferroviária, suporte para tolerância a falhas de hardware e suporte para tempo real.

A implementação do *framework* proposto no Capítulo 5 pode gerar importantes discussões sobre as estruturas de projeto definidas e, conseqüentemente, sobre os padrões de projetos e metapadrões utilizados. O *framework* deve ser implementado tendo em vista a provisão de um ambiente completo para a construção de controladores de trens. Portanto, ênfase deve ser dada à interface com o usuário do *framework*.

Um tópico que pode ser explorado em alguns aspectos do nosso trabalho é reflexão computacional. Reflexão computacional [Mae87] é um mecanismo que permite a um sistema manipular ou modificar seu comportamento devido à incorporação de estruturas

que representam seu próprio estado. Uma arquitetura reflexiva (ou seja, arquitetura que suporta o conceito de reflexão computacional) é dividida em duas partes: o nível base, relacionado com a solução de problemas do domínio da aplicação e a parte de reflexão computacional (meta-nível), relacionada com a solução de problemas e armazenamento de informações sobre o nível base. Nós acreditamos que o mecanismo de reflexão pode ser combinado com o mecanismo de delegação para tornar transparente a implementação da máquina de estados e transições. Além disso, reflexão pode ser utilizada na implementação dos requisitos de distribuição. O componente Protocolo de Comunicação pode ser criado de maneira transparente no meta-nível e apresentado ao nível base como um objeto local, embora ele seja uma referência para um objeto remoto.

Um tópico mais diretamente relacionado com o funcionamento do Controlador de Trens, mas não menos importante, é a implementação de algoritmos de grafos para controlar os trajetos dos trens e as regiões de controle na malha ferroviária. Uma idéia inicial é dividir o problema em dois níveis. No primeiro nível teríamos um grafo representando o movimento do trem pelas seções. Neste caso, não estaríamos preocupados com o comportamento dos conectores, o importante seria reservar uma determinada seção ou um conjunto de seções (conforme o tipo de região de controle desejado). No segundo nível, os conectores seriam tratados como vértices e, com isso, teríamos um grafo mais genérico. O desafio consiste em mapear o comportamento do primeiro nível para o segundo nível, respeitando as restrições definidas no modelo.

Por fim, acreditamos que a nossa abordagem para atender os requisitos de tolerância a falhas de ambiente e distribuição é apropriada também para dar suporte a tolerância a falhas de hardware e tempo real, mas isto também tem que ser demonstrado.

Referências Bibliográficas

[BN84] A. Birrel e B. Nelson. *Implementing Remote Procedure Calls.* ACM Trans. Computer Systems, fevereiro de 1984, p. 39-59.

[Boo91] G. Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings, Menlo Park, CA, 1991.

[BRJ97] G. Booch, J. Rambaugh e I. Jacobson. *Unified Modeling Language.* Versão 1.0, Rational Software Corporation, janeiro de 1997.

[CDK94] G. Coulouris, J. Dollimore e T. Kindberg. *Distributed Systems: Concepts and Design.* Addison-Wesley Publishing Company, segunda edição, 1994.

[CR86] R. H. Campbell e Brian Randel. *Error recovery in asynchronous systems.* IEEE Transactions on Software Engineering, SE-12, número 8, agosto de 1986.

[Cri91] F. Cristian. *Understanding Fault-Tolerant Distributed Systems.* Communications of the ACM, 34 (2), fevereiro de 1991.

[DK96] P. Dagermo e J. Knutsson. *Development of an Object-Oriented Framework for Vessel Control Systems.* Technical Report, ESPRIT III/ESSI/DOVER, Dover Consortium 1996.

[Fir93] D. G. Firesmith. *Frameworks: The golden path to object Nirvana.* Journal of Object-Oriented Programming -JOOP, outubro de 1993.

[GHJV93] E. Gamma, R. Helm, R. Johnson e J. Vlissides. *Design Patterns: Abstraction and Reuse of Object-Oriented Design.* In Proceedings of the ECOOP'93 Conference, Kaiserlautern, Germany, Springer Verlag, 1993.

[GHJV94] E. Gamma, R. Helm, R. Johnson e J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley Publishing, Massachusetts, USA, 1994.

[Hel95] R. Helm. *Patterns in Practice.* Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95), 1995, p. 337-341.

[HW92] W. L. Heimerdinger e C. B. Weinstock. *A Conceptual Framework for System Fault Tolerance.* Technical Report CMU/SEI-92-TR-33, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1992.

[Jac92] I. Jacobson. *Object-Oriented Software Engineering - A Use Case Driven Approach.* Addison-Wesley, 1992.

[Jal94] P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, Inc., Englewood Cliffs, New Jersey, USA, 1994.

[JF88] R. E. Johnson e B. Foote. *Designing Reusable Classes*. Journal of Object-Oriented Programming -JOOP, 1 (2):22-35, Junho/Julho 1988.

[Joh92] R. E. Johnson. *Documenting Frameworks using Patterns*. In Proceedings of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'92), Vancouver, Canada, 1992.

[JZ91] R. E. Johnson e J. M. Zweig. *Delegation in C++*. Journal of Object-Oriented Programming, novembro/dezembro de 1991, 4 (7): 31-34.

[LA90] P. A. Lee e T. Anderson. *Fault Tolerance: Principles and Practice*. Pretince-Hall International, Inc., Englewood Cliffs, NJ, USA, 1990.

[LK94] R. Lajoie e R. K. Keller. *Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts and Motifs in Concert*. In Proceedings of the 62nd Congress of the Association Canadienne Française pour l'Avancement des Sciences (ACFAS), Montreal, Canada, maio de 1994.

[LRH96] G. C. Low, G. Rasmussen e B. Henderson-Sellers. *Incorporation of distributed computing concerns into object-oriented methodologies*. Journal of Object-Oriented Programming, junho de 1996, p. 12-20.

[Mae87] P. Maes. *Concepts and Experiments in Computational Reflexion*. Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), 1987, p. 147-155.

[Mey93] B. Meyer. *Systematic concurrent object-oriented programming*. Communications of the ACM, 36 (9), p. 56-80, setembro de 1993.

[Nie89] O. Nierstrasz. *A Survey of Object-Oriented Concepts*. In Object-Oriented Concepts, Databases and Applications, ed. W. Kim e F. Lochovsky, p. 3-21, ACM Press e Addison-Wesley, 1989.

[Ode94] J. J. Odell. *Six different kinds of composition*. Journal of Object-Oriented Programming -JOOP, janeiro de 1994.

[Pre95] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.

[QR96] E. M. Quadros e C. M. F. Rubira. *Tolerância a Falhas num Controlador de Trens Orientado a Objetos e Distribuído*. I Simpósio Regional de Tolerância a Falhas, Porto Alegre-RS, dezembro de 1996.

[RR95] C. M. F. Rubira e B. Randell. *Object-Oriented Environmental Fault Tolerance*. VI SCTF - Simpósio de Computadores Tolerantes a Falhas, agosto 1995, Canela-RS (Brasil), p. 417-439.

[Rub94] C. M. F. Rubira. *Structuring Fault-Tolerant Object-Oriented Systems Using Inheritance and Delegation*. PhD thesis, Dept. of Computing Science, University of Newcastle upon Tyne, outubro de 1994.

[Rum92] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy e W. Lorensen. *The Object-Oriented Modeling and Design*. Pretince-Hall International, Inc., Englewood Cliffs, NJ, USA, 1992, Segunda Edição.

[Rum95a] J. Rumbaugh. *OMT: The object model*. Journal of Object-Oriented Programming -JOOP, 7 (8), janeiro de 1995.

[Rum95b] J. Rumbaugh. *OMT: The dinamic model*. Journal of Object-Oriented Programming -JOOP, 7 (9), fevereiro de 1995.

[RZ96] D. Riehle e H. Züllighoven. *Understanding and Using Patterns in Software Development*. Theory and Practice of Object Systems 2 (1), 1996.

[SC95] A. Sane e R. Campbell. *Object-Oriented State Machines: Subclassing, Composition, Delegation and Genericity*. Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'95), 1995, p. 17-32.

[Sch95] H. A. Schmid. *Creating the Architecture of a Manufacturing Framework by Design Patterns*. Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95), 1995, p. 370-384.

[Sch96a] H. A. Schmid. *Design patterns for constructing the hot spots of a manufacturing framework*. Journal of Object-Oriented Programming -JOOP, janeiro de 1994.

[Sch96b] H. A. Schmid. *Creating Applications from Components: a Manufacturing Framework Design*. IEEE Software, Volume 13, Número 6, novembro de 1996.

[Sdt95a] D. C. Schmidt. Reactor: *An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching*. In Patterns Languages of Program Design (J. O. Coplien e D. C. Schmidt), Reading, MA: Addison-Wesley, 1995.

[Sdt95b] D. C. Schmidt. *Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software*. Communication of the ACM, 38 (10), outubro 1995.

[SF95] G. Sonnenberger e Hans-Peter Frei. *Design of a Reusable IR Framework*. In Proceedings of the 18th Annual Int. ACM SIGIR Conference on R & D in Information Retrieval (SIGIR '95), p. 49-57, 1995.

[Shr95] S. K. Shrivastava. *Lessons Learned from Building and Using the Arjuna Distributed Programming System*. VI SCTF - Simpósio de Computadores Tolerantes a Falhas, agosto de 1995, Canela-RS (Brasil).

[Ste87] L. A. Stein. *Delegation is Inheritance*. Conference on Object-Oriented Programming: Systems, Languages and Application (OOPSLA'87), Orlando, Florida, Special Issue of ACM SIGPLAN Notices, 22 (12): 138-146, outubro de 1987.

[Str93] B. Stroustrup. *C++: manual de referência comentado*. Editora Campus, Rio de Janeiro, 1993.

[Tal94] Taligent, Inc. *Building Object-Oriented Frameworks*. Taligent White Paper. 1994.

[Tan92] Andrew S. Tanenbaum. *Modern operating systems*. Prentice-Hall, Englewood Cliffs (NJ), 1992.

[WG94] André Weinand e Erich Gamma. *ET++ - a Portable, Homogenous Class Library and Application Framework*. In Proceedings of UBILAB Conference '94, Universitätsverlag Konstanz, 1994.

[WJ90] R. J. Wirfs-Brock e R. E. Johnson. *Surveying current research in object-oriented design*. Communications of the ACM, 33 (9), setembro de 1990.