Construção de um Framework para Sistemas Controladores de Trens utilizando Padrões de Projeto e Metapadrões*

Elbson Moreira Quadros†
Cecília Mary Fischer Rubira‡

Instituto de Computação
Universidade Estadual de Campinas
{elbson, cmrubira}@dcc.unicamp.br

Resumo

Este artigo apresenta o processo de desenvolvimento de um *framework* caixa-branca (que inclui também componentes caixa-preta) para um subdomínio específico de Controladores de Trens. O *framework* é estruturado através de uma seqüência de transformações no modelo básico de um sistema controlador de trens, utilizando-se os conceitos de padrões de projeto e metapadrões. A partir deste modelo básico, 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.

Palavras-chave: orientação a objetos, padrões de projeto, metapadrões.

Abstract

This paper presents the design process of a domain specific white-box framework (with some black-box components) for a Railway Controller System subdomain. The design is performed as a sequence of transformation steps on a railway controller basic model using design patterns and metapatterns. Starting out from such basic model, which does not provide a degree of reusability sufficiently high for a framework, the transformation steps are applied to this basic model aiming at the increase of its reusability degree.

Keywords: object-oriented systems, framework, design patterns, metapatterns.

1. Introdução

Uma das principais vantagens do paradigma de orientação a objetos é o suporte para reutilização de software [WJ90]. Entretanto, têm-se 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

* Trabalho parcialmente financiado pela FAPESP (Processo nº 96/2705-4).
† Mestre em Ciência da Computação, Instituto de Computação — UNICAMP.
‡ Professora do Instituto de Computação da UNICAMP. PhD University of NewCastle upon Tyne.

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 funcionalidade de um *framework* caracteriza-se por pontos fixos, que não podem ser mudados e pontos adaptáveis, que 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 [Sch96a]. 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.

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 através de herança 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 um conjunto de classes alternativas para cada ponto adaptável. 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.

Este artigo 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 básico de um sistema controlador de trens, utilizando-se os conceitos de padrões de projeto e metapadrões. Este modelo básico constitui o resultado de uma modelagem orientada a objetos para uma configuração específica de um controlador de trens. A partir deste modelo, 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 artigo é organizado da seguinte maneira: Seção 2 define os conceitos de padrões de projeto e metapadrões; Seção 3 descreve duas abordagens para o desenvolvimento de *frameworks* baseado em pontos adaptáveis; Seção 4 descreve o modelo básico do sistema controlador de trens; Seção 5 apresenta a descrição do *framework*; Seção 6 apresenta alguns trabalhos relacionados; e, por fim, na Seção 7 apresentamos algumas conclusões.

2. Conceitos Básicos

Padrões de Projeto

Padrões de projeto (em inglês, *Design Patterns*) [GHJV94] têm sido propostos como meio de representar, registrar e reutilizar micro-arquiteturas de projeto repetitivas, bem como a experiência acumulada 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.

Metapadrões

Metapadrões (em inglês, *metapatterns*) representam um abordagem proposta por Wolfgang Pree [Pre95], que consiste na especificação de um conjunto de metapadrões 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 em um metanível. Portanto, metapadrões não substituem as abordagem de padrões de projeto mas complementam-as (...)".

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*. 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 [Pree95].

Pree define um conjunto de sete metapadrões que implementam o comportamento dos métodos componentes e genéricos:

- **metapadrão Unificação** (Figura 1a): a classe genérica e a classe componente são unificadas em uma única classe;
- **metapadrão Conexão 1:1** (Figura 1b): 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;
- **metapadrão Conexão 1:N** (Figura 1c): um objeto de uma classe genérica refere-se a qualquer número de objetos da classe componente;
- **metapadrão Conexão Recursiva 1:1** (Figura 1d): um objeto de uma classe genérica refere-se a exatamente um objeto de sua classe componente. A classe genérica é descendente da classe componente;
- **metapadrão Unificação Recursiva 1:1** (Figura 1e): versão modificada do metapadrão Conexão Recursiva 1:1, onde classes genéricas e classes componentes são unificadas em

uma única classe;

- **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 1f);
- **metapadrão Unificação Recursiva 1:N** (Figura 1g): versão modificada do metapadrão Conexão Recursiva 1:N,. onde classes genérica e classes componentes são unificadas em uma única classe;

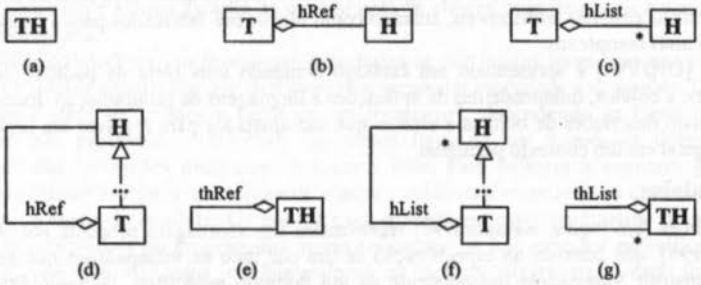

Figura 1 - Conjunto de Metapadrões

3. 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.

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 2.

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.

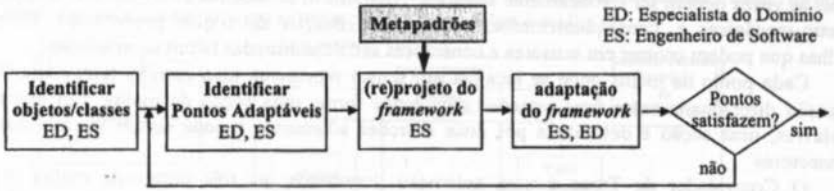

Figura 2 - Abordagem baseada em Pontos Adaptáveis

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 padrões primeiramente, 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*.

4. Modelo Básico

Esta seção descreve as principais características do modelo de análise para uma configuração específica de um Controlador de Trens [Rub94, Qua97]. Este Controlador é um sistema de controle e monitoração de um modelo de ferrovia (Figura 3). Este modelo é composto por uma malha ferroviária e um conjunto de trens. A malha ferroviária 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).

Sensores são dispositivos que detectam a presença de trens e representam a única fonte de informação sobre o estado do sistema. Porém, sensores não são dispositivos confiáveis, pois às vezes podem ser erroneamente ativados. Além disto, conectores estão sujeitos a falhas eletro-mecânicas e, conseqüentemente, trens podem desviar do trajeto predefinido. Estas falhas que podem ocorrer em sensores e conectores são denominadas falhas de ambiente.

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.

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.

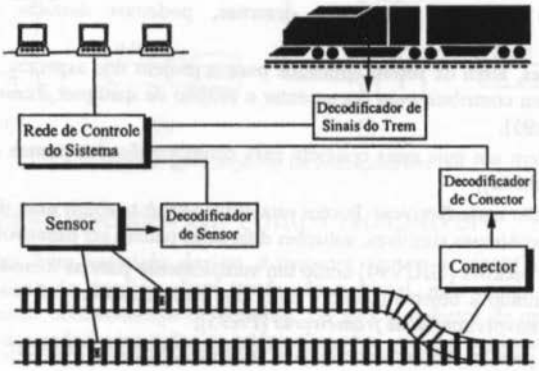

Figura 3 - Componentes do Controlador de Trens

A Figura 4 apresenta (em notação UML [BRJ97]) os principais componentes do diagrama de classes do Controlador de Trens[1]. 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. A classe Hardware da Ferrovia 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. A classe Protocolo de Comunicação 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. A classe Malha generaliza uma série de componentes — sensores, estações e conectores. 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

[1] Maiores detalhes sobre a especificação e o modelo de análise do controlador de trens podem ser encontrados em [Qua97, Capítulo 4].

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. A classe Trem fornece operações para controlar a movimentação dos trens sobre a malha.

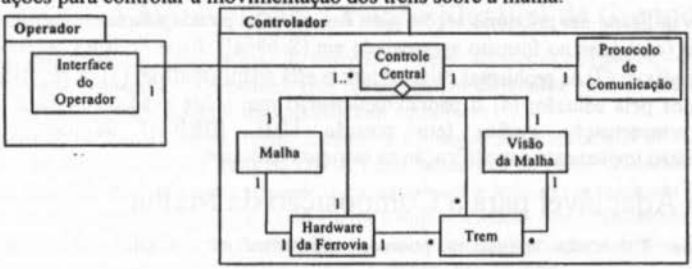

Figura 4 - Principais Componentes do Diagrama de Classes

5. 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 na seção anterior. O principal objetivo 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. 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;

Os pontos adaptáveis do *framework* para Controladores de Trens incluem[2]:

- 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;
- diferentes tipos de componentes: trens, sensores e atuadores. Diferentes aplicações contêm diferentes tipos de trens, sensores e atuadores, os quais apresentam funções similares;
- tolerância a falhas. Flexibilidade para a definição dos estados (normais e anormais) dos componentes tolerantes a falhas.
- Tratamento de eventos. Aplicações específicas podem definir ou redefinir diferentes tipos de tratadores de eventos e/ou eventos;

[2] Uma descrição detalhada dos pontos adaptáveis do *framework* pode ser encontrada em [Qua97, Capítulo 5]. Neste artigo nos limitamos a comentar apenas os principais pontos adaptáveis.

- 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: (1) o ponto adaptável; (2) os problemas relacionados a esta adaptabilidade; (3) os requisitos a serem cumpridos pela solução; (4) o padrão/metapadrão que cobre a adaptabilidade, bem como uma representação gráfica (em notação UML [BRJ97]) de como este padrão/metapadrão implementa a modificação na estrutura existente;

5.1 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 5, 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 5) retorna verdadeiro se o objeto não estiver sendo acessado e falso, caso contrário.

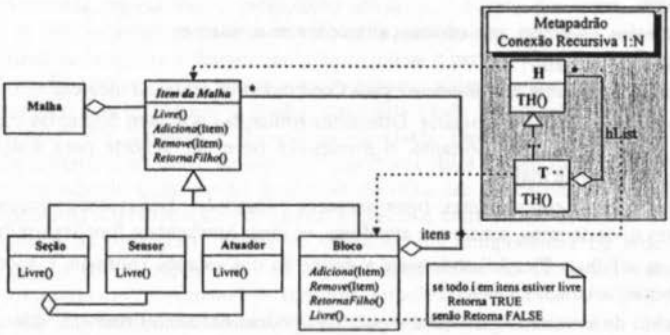

Figura 5 - Estrutura para Composição da Malha Ferroviária

O *metapadrão Conexão Recursiva 1:N* é aplicado à estrutura para composição da malha ferroviária do *framework*. A Figura 5 ilustra como os componentes deste metapadrão correspondem às classes do *framework*.

5.2 Ponto Adaptável para Criação Flexível 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 responsabilidades 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 6, 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.

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[3].

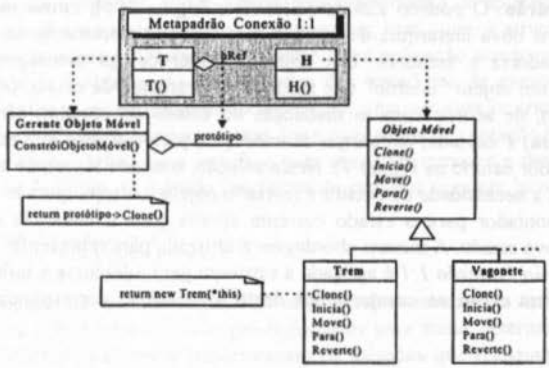

Figura 6 - Criação Flexível de Objetos Móveis

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

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);
```

O mesmo padrão é aplicado para criação flexível dos componentes Atuador e Sensor.

O *metapadrão Conexão 1:1* é aplicado às estruturas dos componentes do *framework* (objetos móveis, atuadores e sensores). A Figura 6 ilustra como os componentes deste metapadrão correspondem às classes do *framework*.

5.3 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 7, permite criar uma nova hierarquia de estados para modelar claramente os estados normal e anormal de atuadores e sensores. Um objeto Atuador delega mensagens para o objeto EstadoAtuador: um objeto "interno" que muda dinamicamente de estado (AtuadorNormal ou AtuadorAnormal), de acordo com as mudanças no estado do atuador. Uma operação (por exemplo, Reserva) 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 Atuador pode mudar sem a necessidade de excluir e recriar o objeto, ou seja, quando um atuador muda de estado, o apontador para o estado corrente aponta para a instância da subclasse que implementa o novo estado. A mesma abordagem é utilizada para representar sensores.

O *metapadrão Conexão 1:1* é aplicado à estrutura para tolerância a falhas do *framework*. A Figura 7 ilustra como os componentes deste metapadrão correspondem às classes do *framework*.

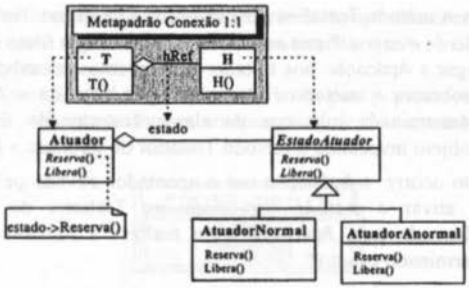

Figura 7 - Estrutura para Tolerância a falhas em Atuadores

5.4 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. Está 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 o acoplamento entre os objetos. Ao invés do objeto *Controle Central* centralizar todo o controle, cada objeto assume suas próprias responsabilidades, tratando os eventos apropriados.

Padrão/metapadrão: O *padrão Reator* [Sdt95a, Sdt95b], como mostrado na Figura 8, 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.

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[4]:

```
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:

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

- 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;

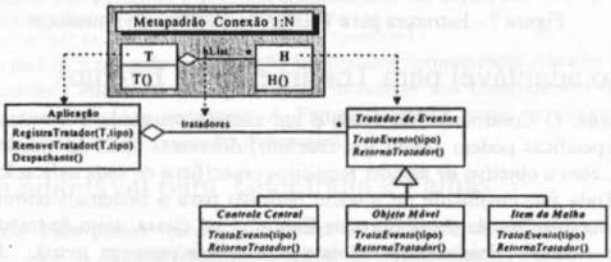

Figura 8 - Estrutura para Tratamento de Eventos

O *metapadrão Conexão 1:N* é aplicado à estrutura de tratamento de eventos do *framework*. A Figura 8 ilustra como os componentes deste metapadrão correspondem às classes do *framework*.

5.5 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 9, 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.

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 acoplado 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 ProtocolocoComunicação, 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 9 ilustra como os componentes deste metapadrão correspondem às classes do *framework*.

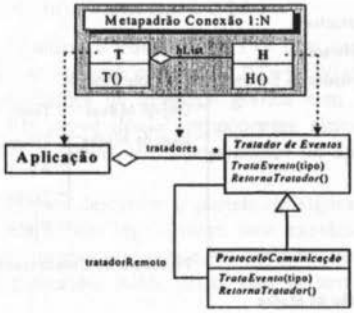

Figura 9 - Estrutura para o Protocolo de Comunicação

Quanto à criação do Protocolo de Comunicação, podemos aplicar o *metapadrão Unificação* para obter uma criação flexível deste objeto (Figura 10). 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*.

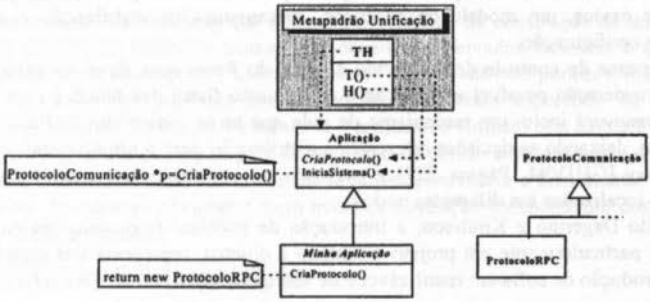

Figura 10 - Criação Flexível do Protocolo de Comunicação

5.6 Estrutura Geral do *Framework*

A Figura 11 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.

Toda aplicação construída a partir do *framework* tem exatamente um objeto da classe Aplicação (Figura 11). 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.

Figura 11 - Estrutura Geral do *Framework*

6. Trabalhos Relacionados

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 que objetos são 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 neste artigo 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 de implementar inicialmente 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 classificar o projeto de um *framework* de maneira eficiente e em um nível de abstração bem mais alto do que em linguagens de programação. Além disso, eles podem ser aplicados em qualquer *framework*.

7. Conclusões

Nós mostramos como um *framework* para um 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 no modelo básico de um Sistema Controlador de Trens, utilizando-se os conceitos de padrões e metapadrões. A partir deste modelo básico, 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 pontos devem ser fixos. Para obter-se um "balanceamento" correto entre os pontos fixos e os pontos adaptáveis requer-se um profundo conhecimento do domínio específico da aplicação e numerosas iterações do projeto do *framework*. Verificamos também que padrões de projeto e metapadrões são técnicas poderosas para se estruturar os pontos adaptáveis de um *framework*. Estas técnicas representam uma excelente maneira de desenvolver e documentar o projeto de um *framework* de maneira eficiente e num nível de abstração independente dos detalhes de linguagens de programação.

8. Referências Bibliográficas

[BRJ97] G. Booch, J. Rumbaugh and I. Jacobson. *Unified Modeling Language*. Version 1.0, Rational Software Corporation, January 1997.

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

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

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

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

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

[LK94] R. Lajoie and 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, May 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.

[Qua97] E. M. Quadros. *Uma Abordagem Orientada a Objetos para Programação Distribuída Confiável*. Tese de Mestrado, Instituto de Computação - UNICAMP,1997.

[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, october 1994.

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

[Sch96a] Hans A. Schmid. *Design patterns for constructing the hot spots of a manufacturing framework*. Journal of Object-Oriented Programming -JOOP, January 1996.

[Sch96b] Hans A. Schmid. *Creating Applications from Components: a Manufacturing Framework Design*. IEEE Software, Volume 13, Number 6, November 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 and 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), October 1995.

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

[WG94] André Weinand and 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 and R. E. Johnson. *Surveying current research in object-oriented design*. Communications of the ACM, 33 (9), September 1990.