Uma Abordagem Reflexiva para a Construção de Frameworks para Interfaces Homem-Computador

Marília Gabriela Coelho

Dissertação de Mestrado

Uma Abordagem Reflexiva para a Construção de Frameworks para Interfaces Homem-Computador

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Marília Gabriela Coelho e aprovada pela Banca Examinadora.

Campinas, 25 de novembro de 1998.

Cecília Mary Fischer Rubira
(Orientadora)

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

Instituto de Computação
Universidade Estadual de Campinas

Uma Abordagem Reflexiva para a Construção de Frameworks para Interfaces Homem-Computador

Marília Gabriela Coelho

Outubro de 1998

Banca Examinadora:

- Profa. Dra. Cecília Mary Fischer Rubira (Orientadora)
- Prof. Dr. Marcos Roberto da Silva Borges

Núcleo de Computação Eletrônica- Instituto de Matemática - UFRJ

- Profa. Dra. Eliane Martins

Instituto de Computação - UNICAMP

- Prof. Dr. Luiz Eduardo Buzato

Instituto de Computação - UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Coelho, Marília Gabriela

C65a Uma abordagem reflexiva para a construção de frameworks para interfaces homem-computador / Marília Gabriela Coelho -- Campinas, [S.P. :s.n.], 1998.

Orientador : Cecília Mary Fischer Rubira

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

1. Interfaces de usuário (Sistema de computador). 2. Frameworks (Programa de computador). I. Rubira, Cecília Mary Fischer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

TERMO DE APROVAÇÃO

Dissertação defendida e aprovada em 25 de novembro de 1998, pela Banca Examinadora composta pelos Professores Doutores:

Prof. Dr. Marcos Roberto da Silva Borges
UFRJ

Profa. Dra. Eliane Martins
IC - UNICAMP

Profa. Dra. Cecília Mary Fischer Rubira
IC - UNICAMP

Para ser grande, Sê inteiro: nada

Teu exagera ou exclui.
Sê todo em cada coisa. Põe quanto és

No mínimo que fazes.
Assim em cada lago a Lua toda

Brilha, porque alta vive.

Fernando Pessoa enquanto
Ricardo Reis

À memória de meu pai João Cloves,
com carinho.

Agradecimentos

Agradeço a

Deus pela força e perseverança a mim concedidas para a realização deste trabalho.

Minha orientadora Cecília Mary Fischer Rubira pela paciência e dedicação.

Toda a minha família e principalmente a minha mãe Norma Célia Coelho e meu irmão George Albert Coelho pelo apoio e por acreditarem na realização deste trabalho.

Os colegas Elbson, Oliva e Cleidson pelas contribuições técnicas.

Os amigos que sempre estiveram presentes neste período da minha vida, principalmente Alessandra, Luciano, Marcelo, Freud, Patrícia, Gisele, Glaucia, Janne, Cristina, Cristiane, Selma, Amanda, Mário, Laura, Edicézar, Márcio, Cleidson, Ralph, Paola, Juliana, Gutemberg, André, Rogério, Emerson, Pavão, Delano, Marcos André, Roseli e Vera.

Os coros Oficina Coral e Cais do Canto pela oportunidade de aprender um pouco de música e por tornar meus dias mais leves e descontraídos, neste período, através do canto.

A instituição de fomento à pesquisa CAPES pela ajuda financeira.

Resumo

Uma aplicação interativa é composta de um núcleo funcional (ou aplicação) e da interface do usuário. Uma das maiores dificuldades no desenvolvimento e manutenção de aplicações interativas encontra-se nas dependências entre os módulos que implementam o núcleo funcional e a interface do usuário. Os *frameworks* atuais para construção de aplicações interativas são baseados em padrões de arquitetura, que não provêem independência de diálogo efetiva para a aplicação. Isto acontece porque, nos padrões de arquitetura existentes, as dependências entre os módulos de uma aplicação interativa são implementadas nos próprios módulos. O objetivo deste trabalho é mostrar como o padrão de arquitetura reflexão computacional pode ser utilizado para implementar de forma efetiva a independência de diálogo em aplicações interativas. Nesta abordagem, os serviços de visualização da interface são fornecidos de forma não intrusiva e transparente para a aplicação, pois as dependências entre aplicação e interface são implementadas no meta-nível, independentemente do núcleo funcional da aplicação. Este trabalho propõe o padrão de arquitetura MVCR (Modelo Visão Controlador Reflexivo) como uma combinação do padrão de arquitetura para construção de sistemas interativos MVC (*Model View Controller*) e o padrão de arquitetura reflexão computacional. O padrão MVCR é comparado ao padrão MVC através da efetuação de um estudo de caso utilizando o ET++, um *framework* para desenvolvimento de aplicações gráficas, cuja arquitetura é baseada no MVC. Para realizarmos o estudo de caso, re-estruturamos o ET++ utilizando o padrão MVCR (obtendo-se assim o ET++ Reflexivo) e desenvolvemos uma mesma aplicação utilizando as duas abordagens: o ET++ original baseado no MVC e o ET++ Reflexivo baseado no MVCR. Desta forma, foi possível comparar e avaliar as vantagens e limitações da abordagem reflexiva.

Palavras-chave: *frameworks* para interfaces gráficas, reflexão computacional, MVC, *framework* ET++.

Abstract

An interactive application is compound by a functional core application (or just application) and its graphical user interface (GUI). One of the main difficulties on the development and maintenance of interactive applications is on the dependence between the application and its graphical user interface. Most of the object-oriented GUI frameworks are based on arquitectural patterns, that do not provide dialog independence among applications, this occurs in these patterns because the dependence between the application components and GUI components is implemented in the own components. The goal of this work is to show how computational reflection can be used to implement an effective dialog independence in interactive applications. In this appoach, the graphical user interface can be provided in a transparent and non-intrusive way to the application, since the dependence between the application and GUI are implemented on the meta-level. This work proposes the arquitectural pattern RMVC (Reflective Model/View/Controller) – a combination of the arquitectural pattern MVC (Model/View/Controller) and the arquitectural pattern Computational Reflection. We have carried out a case study in order do compare both patterns MVC and MVCR using the ET++, a GUI framework based on pattern MVC. First, we develop the Reflective ET++, restructuring the ET++ using the MVCR, and then we develop one same application using the two approaches: the original ET++ based on MVC and the Reflective ET++ based on MVCR. Thus, it was possible to discuss the advantages and limitations of using the reflective approach.

Palavras-chave: GUI framework, computational reflection, MVC, framework ET++.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Com o advento da orientação a objetos e o surgimento das ferramentas para o desenvolvimento de interfaces homem-computador, as aplicações interativas têm se mostrado cada vez mais amigáveis[1] para o usuário. Uma aplicação interativa é composta de um núcleo funcional (ou aplicação) e da interface homem-computador (*GUI- Grafical User Interface*) [HHa93]. Apesar da atenção destinada ao *look and feel* das interfaces gráficas, fala-se pouco sobre técnicas e métodos para o desenvolvimento da interface gráfica como um módulo que deve interagir com o módulo da aplicação. Neste sentido foram criados padrões de arquitetura que visam modularizar o sistema, como o MVC (*Model/View/ Controller*)[Gold83] por exemplo, e foram desenvolvidos *frameworks* orientados a objetos para interfaces homem-computador baseados nesses padrões. O MVC distribui o gerenciamento da interface gráfica sobre três componentes: Modelo, Visão e Controlador. O componente Modelo denota qualquer objeto da aplicação; o componente Visão contém os objetos gráficos pertencentes à interface; e o componente Controlador controla a interação com o usuário. *Frameworks* orientados a objetos, segundo Firesmith [Fir93], são coleções de classes colaborativas que capturam padrões e mecanismos que implementam os requisitos comuns de projeto em um domínio específico de aplicação. O domínio de aplicações dos *frameworks* para interfaces homem-computador são os sistemas interativos.

Embora os padrões de arquitetura modularizem o sistema interativo em componentes bem definidos, eles permitem que as dependências entre os componentes sejam implementadas no próprio componente. Principalmente os aspectos de controle da aplicação, que implementam o protocolo de comunicação entre os objetos da visão e do modelo, não são separados do núcleo funcional da aplicação. Esta característica dos padrões de arquitetura para *frameworks* para interfaces homem-computador fere o conceito de independência de diálogo introduzido por Hartson e Hix [Hhi89], que diz que o software da aplicação deve ser desenvolvido independentemente do software da interface. O forte acoplamento entre o núcleo funcional da aplicação e as dependências com a interface gráfica dificultam e, em muitos casos, impedem que a aplicação ou interface sejam reutilizadas independentemente. A adição de uma interface gráfica a uma aplicação já

[1] Do inglês *user friendly*.

implementada, por exemplo, torna-se uma tarefa extremamente trabalhosa, pois muitas alterações deverão ser feitas nos códigos fonte da aplicação.

O objetivo deste trabalho é mostrar como reflexão computacional pode ser utilizada para implementar a independência de diálogo em *frameworks* para interfaces homem-computador no meta-nível. Obtém-se uma independência de diálogo efetiva quando os serviços de visualização da interface gráfica são fornecidos de forma não intrusiva e transparente para a aplicação. Reflexão computacional é um padrão de arquitetura que proporciona uma nova dimensão de modularidade no desenvolvimento de software - a separação entre o nível base e meta-nível de um software. Neste trabalho, definimos o MVCR (Modelo Visão Controlador Reflexivo), como uma combinação do padrão de arquitetura MVC com o padrão de arquitetura reflexão computacional. O MVCR provê a divisão de um sistema interativo em componentes bem definidos (Modelo Visão e Controlador), completamente independentes devido a separação das dependências entre eles, as quais são implementadas no meta-nível.

O padrão MVCR é comparado ao padrão MVC através da efetuação de um estudo de caso utilizando o ET++, um *framework* para desenvolvimento de aplicações gráficas, cuja arquitetura é baseada no MVC. Para realizarmos o estudo de caso, re-estruturamos o ET++ utilizando o padrão MVCR (obtendo-se assim o ET++ Reflexivo) e desenvolvemos uma mesma aplicação utilizando as duas abordagens: o ET++ original baseado no MVC e o ET++ Reflexivo baseado no MVCR. Desta forma, foi possível comparar e avaliar as vantagens e limitações da abordagem reflexiva.

As principais contribuições deste trabalho são: (i) proposta de um padrão de arquitetura para desenvolvimento de *frameworks* para interfaces homem-computador – o MVCR – como uma combinação dos padrões de arquitetura MVC e Reflexão Computacional; (ii) a implementação de um protótipo do ET++ Reflexivo, uma reestruturação do ET++ baseada no MVCR; e (iii) o estudo comparativo entre o MVC e o MVCR, através de uma mesma aplicação desenvolvida a partir do ET++ e do ET++ Reflexivo, com o objetivo de avaliar as vantagens e desvantagens do MVCR em relação ao MVC.

O restante deste trabalho está dividido da seguinte forma. O capítulo 2 define conceitos básicos de orientação a objetos, descreve notações da UML (Linguagem de Modelagem Unificada), linguagem utilizada para representar os modelos de objetos neste texto e define padrões de software enfatizando os padrões de projetos e padrões de arquitetura. O capítulo 3 discute os *frameworks* para interfaces homem-computador e padrões de arquitetura utilizados em suas construções. O capítulo 4 propõe o padrão de arquitetura MVCR. O capítulo 5 faz um estudo comparativo entre o MVCR e o MVC utilizando o *framework* ET++. Finalmente, o capítulo 6 descreve as conclusões deste trabalho e discute alguns trabalhos futuros.

Capítulo 2

Fundamentos de Orientação a Objetos

Este capítulo descreve alguns conceitos fundamentais da orientação a objetos que são importantes para o entendimento das noções apresentadas nos capítulos seguintes. A Seção 2.1 descreve conceitos básicos da orientação a objetos. A Seção 2.2 define padrões de software como uma forma de reutilizar soluções, elaboradas por projetistas experientes, para problemas comuns de projeto e implementação de sistemas. Padrões de software se dividem em Idiomas, Padrões de Projeto e Padrões de Arquitetura. A Seção 2.3 descreve detalhadamente um padrão de projeto freqüentemente utilizado para implementar o MVC nos sistemas interativos - o padrão de projeto Observador. A Seção 2.4 descreve Metapadrões como uma generalização de padrões de projeto. Finalmente, a Seção 2.5 descreve reflexão computacional como um padrão de arquitetura de importância fundamental para este trabalho.

2.1 O Modelo de Objetos

A Programação Orientada a Objetos usa um modelo de implementação no qual programas são organizados como coleções cooperativas de objetos, os quais representam instâncias de classes em uma hierarquia unidas pelo relacionamento de herança [Boo94]. Esta Seção define conceitos básicos de orientação a objetos segundo Rumbaugh [Rumb91] e Pree[Pree94].

2.1.1 Objetos

Em um sistema, os dados são quantificados em entidades discretas e distinguíveis chamadas objetos. Objetos podem representar entidades concretas do mundo real como um arquivo, por exemplo, ou entidades abstratas como uma política de escalonamento. Objetos possuem a propriedade de **encapsulamento**. Encapsulamento consiste da separação de

aspectos externos de um objeto, que são acessíveis por outros objetos, de detalhes de implementação do objeto, que são escondidos de outros objetos.

2.1.2 Classes

A maioria das linguagens de programação orientadas a objetos utilizam uma classe para descrever o esquema de um objeto. Objetos são instâncias de classes. Objetos da mesma classe possuem a mesma estrutura de dados (**atributos**) e comportamento (operações ou **métodos**). Cada instância tem seu próprio valor para cada atributo, mas compartilha os mesmos nomes de atributos e métodos com os outros objetos da mesma classe. Uma classe pode ser abstrata ou concreta:

- **Classe Abstrata** – uma classe abstrata não possui instâncias diretas, ela define o comportamento comum às suas subclasses. Alguns de seus métodos podem ser implementados enquanto outros podem possuir implementações vazias.

- **Classe Concreta** – possui instâncias direta e todos os seus métodos devem possuir implementação.

2.1.3 Herança

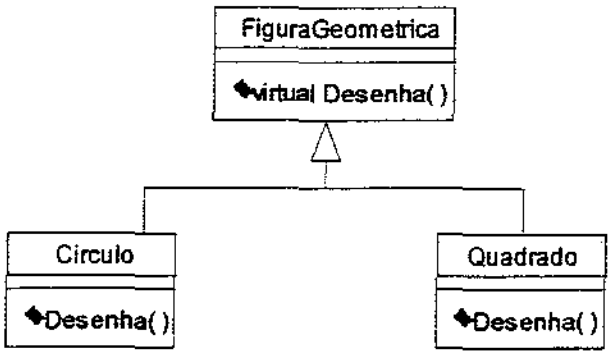
O relacionamento de herança permite o compartilhamento de atributos e métodos entre classes. Se uma classe B herda de uma classe A, todos os atributos e métodos de A passam a fazer parte de B. Diz-se então que a classe B é subclasse da classe A. O comportamento de B é idêntico ao comportamento da classe A, entretanto, a classe B pode modificar o comportamento herdado de A, através da sobrecarga de métodos de A. Um método de A é sobrecarregado na classe B, se ele possui a mesma interface (nome e parâmetros) e uma outra implementação. Além dos métodos e atributos herdados, a classe B pode possuir seus próprios atributos e métodos.

2.1.4 Polimorfismo

Polimorfismo significa que a mesma operação pode comportar-se diferentemente em diferentes classes. Por exemplo, suponha a classe *FiguraGeometrica* da Figura 2.1 que possui a operação *Desenha()*, suponha também que a referida classe possua duas subclasses: *Circulo* e *Quadrado*. Cada subclasse redefine a mesma operação *Desenha()* de acordo com as características gráficas do objeto representado pela classe.

A classe *FiguraGeométrica* implementa as características comuns das figuras permitindo que outras classes, que se relacionam com as figuras geométricas, possam fazê-lo independentemente das características específicas das figuras. O método *Desenha()* da

classe *FiguraGeometrica* representa uma abstração dos métodos *Desenha()* de suas subclasses.



- privado
- público
- protegido

Figura 2.4 : Exemplo de Polimorfismo.

2.1.5 Amarração Dinâmica

Amarração dinâmica significa que os tipos de todas as variáveis e expressões somente são conhecidos em tempo de execução. Algumas linguagens possuem amarração dinâmica como C++, por exemplo. Observe o trecho de código abaixo escrito em C++ referente ao modelo da Figura 2.1:

```
fig1 *FiguraGeometrica;
fig1 = new Circulo();
fig1->Desenha();
```

O tipo estático da variável *fig1* é um apontador para a classe *FiguraGeometrica*. O tipo dinâmico de *fig1* é um apontador para a classe Circulo. Como C++ possui amarração dinâmica, o método *Desenha()* da classe *Circulo* será executado, pois a determinação do método *Desenha()* é feita em tempo de execução e depende do tipo dinâmico do objeto *fig1*.

Se uma linguagem não possui amarração dinâmica, ela possui **amarração estática**. Isso significa que os tipos de todas as variáveis e expressões são conhecidos em tempo de compilação.

Para a utilização de todo potencial do modelo de objetos a fim de construir arquiteturas reutilizáveis, é necessária uma apropriada combinação de conceitos como definição de

objetos e classes e a utilização de herança juntamente com polimorfismo e amarração dinâmica.

2.2 Padrões de Software

Segundo Critopher Alexander [Ale79], um padrão[2] descreve um problema, que ocorre freqüentemente em algum domínio, e a solução para tal problema de forma que possa ser utilizada muitas vezes sem ter de ser elaborada novamente. Ele criou a idéia de linguagem de padrões na arquitetura, a fim de transmitir sua experiência capacitando as pessoas para projetarem suas casas e comunidades [AIS77]. Seus padrões variavam em escalas de problemas que eles solucionavam. Havia padrões que explicavam como quebrar o mundo em nações e as nações em regiões menores, como organizar rodovias, *shoppings*, casas e até os móveis dentro da casa, ou ainda o material a ser usado para construir as paredes. Semelhantemente, o conceito foi levado para o desenvolvimento de software. Neste contexto, os padrões capturam experiências de desenvolvedores em soluções testadas e bem elaboradas que podem ser reutilizadas por outros projetistas.

2.2.1 O que são Padrões de Software?

Cada padrão lida com um problema específico recorrente no projeto ou implementação de um software. Da mesma forma que na arquitetura, pode-se identificar escalas diferentes de problemas a serem solucionados por distintos padrões. Mais especificamente, existem diferentes escalas de abstração de padrões. Alguns padrões contribuem na reestruturação de sistemas de software em subsistemas, outros ajudam no projeto dos subsistemas e componentes e relacionamento entre eles. Há ainda os padrões que ajudam na implementação de aspectos particulares de projeto em uma linguagem específica.

Um padrão pode ser descrito por quatro elementos [GHJ95]:

1. **Nome**: nome através do qual fica sendo conhecida uma solução para determinado problema de projeto. Nomeando padrões aumentamos nosso vocabulário de projeto.
2. **Problema**: descreve quando utilizar o padrão, especificando o problema e seu contexto, ou seja, as situações onde ele ocorre. Além da especificação do problema, Cristopher Alexander descreveu, em alguns de seus padrões, uma série de forças[3] relativas à solução do problema. Essas forças denotavam aspectos que deveriam ser considerados

[2] Do inglês *pattern*.

[3] Do inglês *force*.

na solução do problema, tais como: requisitos que deveriam ser satisfeitos, restrições que deveriam ser consideradas e propriedades desejáveis que a solução deveria possuir. Descrever o contexto significa descrever situações onde o problema ocorre. Procura-se descrever as situações de uso conhecidas para o padrão, visto que a descrição de todas as situações possíveis é impraticável. O contexto serve como um guia apenas, existindo sempre a possibilidade da utilização do padrão em algum novo contexto.

3. **Solução**: descreve os elementos que fazem parte da solução proposta pelo padrão e seus relacionamentos (comportamento estático) e a colaboração desses elementos (comportamento dinâmico). A solução do padrão é, de certa forma, abstrata, visto que deve satisfazer uma família de problemas descritos no elemento problema.
4. **Conseqüências**: são os resultados ou o impacto da aplicação do padrão no sistema. Este item permite a avaliação das alternativas de projeto, pois descrevem os benefícios e custos da aplicação do padrão. Em geral, são feitas considerações a respeito do impacto do padrão na flexibilidade, extensibilidade, e portabilidade do sistema.

Dentre as vantagens da utilização de padrões pode-se citar:

- Reutilização de experiência - os padrões buscam resgatar, enfatizar e documentar soluções que realmente funcionam, soluções estas que só podem ser efetivamente validadas pela experiência e não por testes das mesmas [Sch96]. Um projetista que está familiarizado com tais padrões pode aplicá-los imediatamente em algum projeto sem ter de recriá-los.
- Documentação de sistemas em geral - padrões são considerados uma linguagem de projeto de mais alto nível e mais precisa para documentar sistemas. Um projetista pode dizer, por exemplo, que, para resolver determinado problema, ele utilizou o padrão específico, sem precisar descrever detalhes dos procedimentos. Para tanto, é necessário que os usuários da documentação estejam familiarizados com o padrão citado.
- Documentação de frameworks - Em uma situação mais especifica que a documentação de sistemas em geral, a documentação de *frameworks* possui três propósitos com níveis distintos de detalhamento: descrever o propósito do *framework*, descrever como utilizar o *framework* e descrever o projeto detalhado do *framework*. Padrões são bastante úteis para documentar a forma de utilização e o projeto detalhado do *framework*.
- Facilidade de comunicação entre a equipe de desenvolvimento - Um dos maiores benefícios dos padrões é gerar vocabulário e formas de expressão comuns.
- Suporte à construção de software - os padrões de arquiteturas ajudam no projeto de sistemas em geral facilitando a implementação de características predefinidas, os padrões de projeto auxiliam nos projeto dos subsistemas. Os padrões, em geral, provêem um esqueleto para o comportamento funcional, ajudando na implementação da funcionalidade da aplicação. Por exemplo: existem padrões para manter consistência

entre componentes que trabalham cooperando no fornecimento de algum serviço. Além disso, os padrões dão suporte a requisitos não funcionais para os sistemas de software, tais como confiabilidade, facilidade de alteração, teste e reutilização. Como exemplo de padrão que dá suporte à facilidade de alteração e reutilização do sistema de software pode-se citar o MVC. Cada padrão provê um conjunto predefinido de componentes, regras e relacionamentos entre eles. Em específico, os padrões de projeto funcionam como pequenos blocos que são úteis na construção do sistema. Neste caso, a utilização de padrões aceleram o processo de desenvolvimento e melhoram a qualidade dos sistemas desenvolvidos.

Embora os padrões determinem a estrutura básica da solução para um problema de projeto particular, eles, muitas vezes, não cobrem todas as necessidades específicas do problema que o projetista pretende resolver. Isso acontece porque um padrão provê um esquema para uma solução genérica para uma família de problemas, assim, um padrão não é somente um bloco pré fabricado para ser prontamente utilizado. O projetista precisa adaptar o padrão de acordo com as necessidades específica do problema que se pretende solucionar. Padrões auxiliam na resolução de problemas, mas eles não provêem soluções completas.

Visando formalizar as diferenças entre os níveis de abstração dos padrões, Bushmann et. al. [BMR96] classificaram os padrões de software em 3 categorias que serão discutidas abaixo: **padrões de arquitetura, padrões de projeto** e **idiomas**. Padrões de arquitetura podem ser usados no início do projeto, para estruturar o sistema. Os padrões de projeto são utilizados durante toda a fase de projeto e idiomas, durante a fase de implementação.

2.2.2 Padrões de arquitetura

Padrões de arquitetura descrevem os princípios gerais de estruturação de sistemas de software. Eles especificam as propriedades arquiteturais de uma aplicação, impactando na arquitetura de seus subsistemas. Em suma, pode-se dizer que um padrão de arquitetura expressa um esquema de organização estrutural para sistemas de software. Ele provê um conjunto de subsistemas pré-definidos, especifica responsabilidades desses subsistemas e inclui regras e guias para organizar o relacionamento entre eles [BMR96]. Um exemplo de padrão de arquitetura é o Model/View/Controller (MVC) (descrito na Seção 3.1), um padrão para a estruturação de sistemas de software interativos. Um outro exemplo é o padrão de arquitetura reflexão computacional utilizado para implementar sistemas flexíveis a auto-modificáveis (descrito na Seção 2.5).

2.2.3 Padrões de Projeto

Um padrão de projeto pode ser considerado um conjunto de regras que descrevem como realizar uma determinada tarefa no processo de desenvolvimento de software de subsistemas e componentes. Ele descreve uma estrutura de componentes que soluciona um problema de projeto geral em um contexto particular. Padrões de projeto, são elementos de menor escala que padrões de arquitetura. Diferentemente dos padrões de arquitetura, a utilização de um padrão de projeto não afeta a estrutura fundamental do sistema. Estes têm uma forte influência na arquitetura dos subsistemas e por isso são considerados padrões de média escala.

Gama et al. [GHJ95] catalogaram 23 padrões de projetos, classificando-os de acordo com as características do sistema afetadas pelas suas utilizações. O conjunto de padrões foram divididos em padrões de criação, padrões de estrutura e padrões de comportamento. Os padrões de criação tratam do processo de criação de objetos. Um exemplo é o *Factory Method* que define uma interface para criar objetos, porém deixa a subclasse decidir qual classe instanciar. O *Factory Method* permite uma classe deferir a instanciação de seus objetos para subclasses. Os padrões de estrutura lidam com a composição de classes ou objetos. Um exemplo de padrão de estrutura é o *Composite,* o qual compõe objetos em uma estrutura de árvores para representar hierarquias parte-todo. O *Composite* possibilita que objetos se relacionem com objetos individuais e composição de objetos uniformemente. Padrões comportamentais caracterizam a forma como as classes ou objetos interagem e distribuem responsabilidades. Exemplo de padrão de comportamento é o Observador também conhecido como *Publisher/Subscriber,* utilizado para implementar o mecanismo de propagação de alteração entre objetos dependentes. A Seção 2.3 mostra em detalhes o padrão Observador devido ao fato desse padrão ser utilizado para projetar o MVC nos sistemas interativos.

2.2.4 Idiomas

Idiomas endereçam aspectos particulares de ambos: projeto e implementação. Idiomas são padrões de baixo nível cujo propósito é resolver algum problema de projeto ou implementação em uma linguagem de programação específica. Um Idioma descreve como implementar aspectos particulares de componentes ou relacionamentos entre eles, usando características particulares de uma dada linguagem. Um exemplo de Idioma é o *reference counting* [Cop92] utilizado para gerenciar recursos dinamicamente alocados em C++.

2.3 O Padrão de Projeto Observador

Esta seção descreve o padrão de projeto Observador ou *Publisher-Subscriber* catalogado em [GHJ94]. O padrão de projeto Observador ajuda a manter sincronizados o estado de componentes cooperativos. Para tanto, ele provê um mecanismo unidirecional para propagação de alterações: um objeto observado (*publisher*) notifica qualquer número de objetos observadores (*subscribers*) sobre as mudanças no seu estado, assim que elas ocorrerem [BMR96]. O padrão será descrito nas Subseções seguintes segundo seus elementos básicos.

2.3.1 Problema

Freqüentemente, precisamos implementar dependência entre objetos. Um objeto Ob1 depende de outro objeto Ob2, se o seu estado se altera caso o estado do objeto Ob2 seja alterado. Esse problema pode ser resolvido, introduzindo dependências diretas entre os objetos, entretanto esta solução não é flexível e não permite reutilização dos objetos separadamente. Seria interessante que fosse utilizado um mecanismo mais geral, que pudesse ser aplicado a vários contextos.

A solução deve garantir um mecanismo que permita um objeto possuir vários objetos dependentes; o número de objetos dependentes não precisa ser conhecido *a priori* e pode mudar com o tempo; e além disso a informação entre Observadores e Observados não deve ser fortemente acopladas.

2.3.2 Solução

A Figura 2.2 mostra a arquitetura do padrão. O funcionamento e estrutura do padrão são descritos de forma detalhada no próximo capítulo, onde o padrão é utilizado para implementar as dependências entre aplicação e interface gráfica.

O padrão de projeto oferece as seguintes possibilidades em sua implementação:

- O observado pode decidir quando notificar os seus observadores. Ele pode também alimentar uma fila com várias alterações antes de notificar os observadores.
- Um objeto pode observar vários outros.
- Um objeto pode ser ao mesmo tempo observador e observado.
- A subscrição de um objeto como um observador pode ser diferenciada da garantia de notificação. A notificação pode ser feita aos observadores de acordo com o tipo de evento. Um observador não precisa ser notificado de eventos que não lhe interessam.
- O observado pode enviar detalhes dos dados que foram alterados para os observadores no momento da notificação, ou pode simplesmente notificar e deixar que os observadores encontrem os dados alterados.

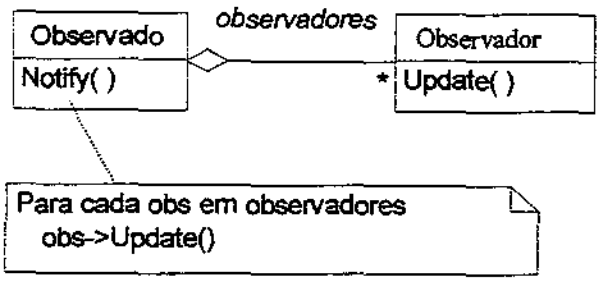


Figura 2.5: Padrão de projeto Observador

2.3.3 Conseqüência

As conseqüências da utilização do padrão seguem abaixo [GHJ94]:

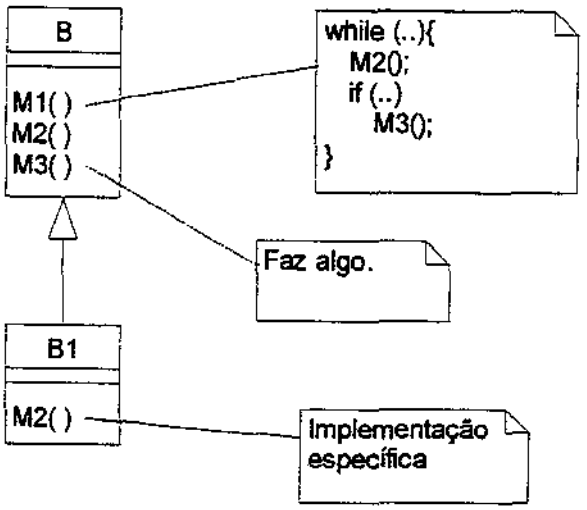
1. Alterações inesperadas - uma operação inócua no objeto observado pode causar uma cascata de alterações para todos os observadores e seus objetos dependentes. Este problema é agravado pelo fato de que um simples protocolo alterado não provê detalhes sobre o que foi alterado no objeto observado. Sem um protocolo adicional, os observadores precisam deduzir as mudanças.
2. Acoplamento abstrato entre observado e observador. O observado não conhece nenhuma classe concreta de qualquer observador.
3. Suporte para a comunicação *broadcast*. A notificação que um objeto observado envia não precisa especificar seu recebedor. A notificação é enviada automaticamente para todos os objetos interessados.

A próxima Seção descreve metapadrões como uma generalização de padrões de projeto. Metapadrões e padrões de projeto são utilizados no Capítulo 5 para descrever o *framework* ET++, utilizado no experimento prático deste trabalho.

2.4 Metapadrões

Metapadrões, segundo Pree [Pree94] são padrões de projeto mais abstratos que os padrões catalogados por Gama [GHJ94]. Os metapadrões permitem categorizar os padrões de projeto descrevendo-os em um meta-nível. Assim, um metapadrão pode ser aplicado a diversas situações distintas nas quais se aplicariam distintos padrões de projeto da mesma categoria que o metapadrão representa. Eles podem ser utilizados para implementar

software genérico orientado a objetos em geral. Segundo Pree, metapadrões não pretendem substituir os padrões de projeto e sim complementá-los [Pree94].



M1 é o método template e seus hooks são os métodos M2 e M3.

Neste exemplo, M2 é um método abstrato e M3 pode ser sobrecarregado, mas possui uma implementação *default* em B.

Figura 2.6: Exemplo de método hook e template [Pree94].

Metapadrões são baseados nos conceitos de **métodos hook** e **métodos template**. Métodos *template* implementam partes fixas de um software genérico. Métodos *hook* implementam as partes que devem ser deixadas flexíveis para serem adaptadas para uma situação específica através de amarração dinâmica. Métodos *template* são baseados em métodos *hook*. *Hooks* e *template* podem não pertencerem a uma mesma classe. A classe que possui o método *template* é chamada classe *template* e a classe que possui o método *hook* é chamada classe *hook*. A Figura 2.3 mostra um exemplo de adaptação de métodos *hooks* em subclasses. A Figura 2.4 mostra graficamente as partes fixas e flexíveis do exemplo da Figura 2.3. A área em branco da figura representa as partes fixas do software e as pequenas áreas em cinza, as parte flexíveis que foram adaptadas pela classe B1 e pelo comportamento *default* de M3 na própria classe B.

As partes em cinza representam os pontos flexíveis que devem ser implementados de acordo com a situação específica de uso do metapadrão.

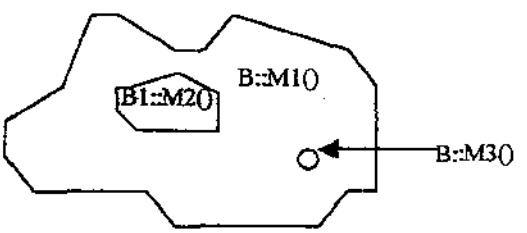


Figura 2.7: Software genérico com pontos adaptáveis.

Pree catalogou sete metapadrões em [Pree94], como mostra a Figura 2.5:

- **Metapadrão Unificação** - a classe template e classe hook são unificadas em uma única classe (Figura 2.5a);
- **Metapadrão Conexão 1:1** - um objeto de uma classe template relaciona-se com um objeto da classe hook (Figura 2.5b);
- **Metapadrão Conexão 1:N** - um objeto de uma classe template relaciona-se com zero ou mais objetos da classe hook (Figura 2.5c);
- **Metapadrão Conexão Recursiva 1:1** - um objeto de uma classe template relaciona-se com um objeto da classe hook. A classe template é descendente da classe hook (Figura 2.5d);
- **Metapadrão Unificação Recursiva 1:1** - modificação do metapadrão conexão recursiva 1:1 em que classe template e classe hook são unificadas (Figura 2.5e);
- **Metapadrão Conexão Recursiva 1:N** - metapadrão semelhante ao metapadrão conexão recursiva 1:1. A diferença está na cardinalidade do relacionamento (Figura 2.5f);
- **Metapadrão Unificação Recursiva 1:N** - difere do metapadrão unificação recursiva 1:1 na cardinalidade do relacionamento (Figura 2.5g).

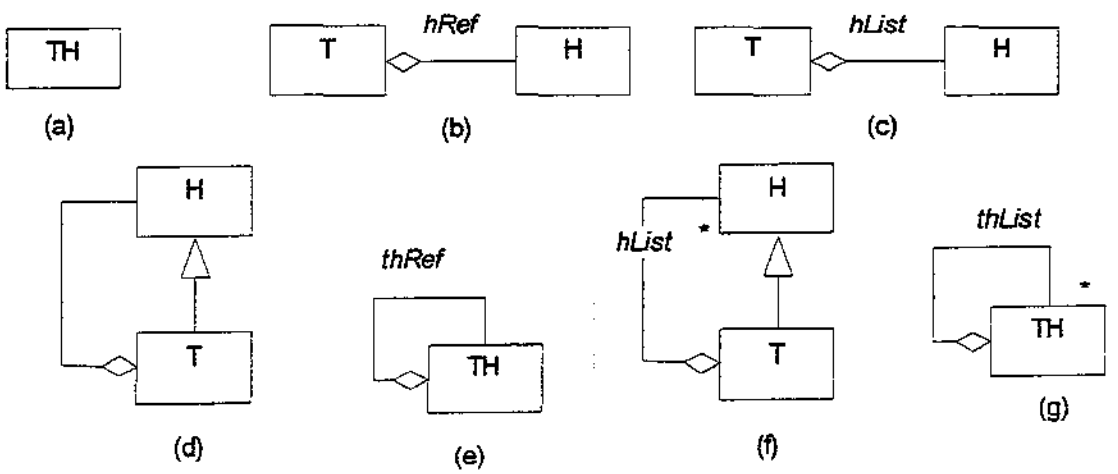


Figura 2.8 : Os sete metapadrões catalogados por Wolfgan Pree.

A Figura 2.5 mostra o **Metapadrão Conexão 1:N** como uma generalização do padrão de projeto **Observador**, descrito na Seção anterior. O metapadrão é considerado uma generalização do Observador porque ele pode ser utilizado em outros contextos também, e não somente para implementar a propagação de alterações entre objetos dependentes.

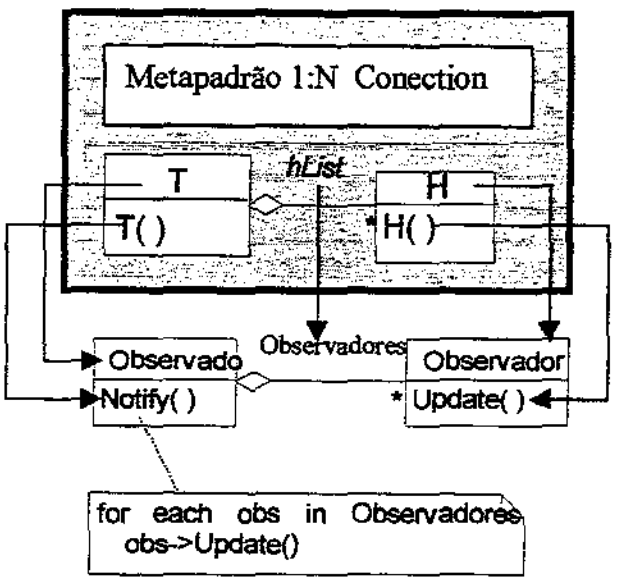


Figura 2.9: O metapadrão 1:N Conection como uma generalização do padrão de projeto Observador.

A próxima Seção descreve o padrão de arquitetura Reflexão Computacional utilizado neste trabalho na elaboração do padrão de arquitetura MVCR (Modelo Visão Controlador Reflexivo), descrito no capítulo 4.

2.5 O Padrão de Arquitetura Reflexão Computacional

Reflexão Computacional segundo Maes [Mae87] é o comportamento exibido por sistemas reflexivos. Um sistema reflexivo é um sistema computacional, que realiza computações sobre objetos que representam as entidades do próprio sistema. Os dados do sistema, representados por esses objetos, devem estar conectados de forma causal ao comportamento do sistema e vice e versa, ou seja, qualquer alteração em um deve ser refletida no outro. É possível fazer acesso e manipular tais objetos utilizando uma outra dimensão de programação: a programação no meta-nível. Dessa forma, o sistema pode raciocinar e agir sobre si próprio.

Esta seção descreve reflexão computacional do ponto de vista de um padrão arquitetural [BMR96] utilizado para proporcionar flexibilidade aos sistemas, de modo que suas arquiteturas possam ser abertas e auto-modificáveis. Esta Seção descreve o padrão de arquitetura de acordo com os elementos: problema, solução e usos conhecidos.

2.5.1 Problema

Softwares evoluem com o tempo. O desenvolvimento das tecnologias exigem que os sistemas incorporem novas características freqüentemente. Sistemas abertos incorporam essas características mais facilmente. Várias forças influenciam este problema:

- Alteração de software é cara, tediosa e extremamente sujeita a erros, visto que uma mudança em um componente do sistema pode afetar outros componentes. Assim, softwares que dão suporte às suas próprias modificações devem fazê-lo de forma efetiva e segura.
- Sistemas de software adaptáveis normalmente possuem uma estrutura interna complexa. Aspectos que estão sujeitos a alterações devem ser encapsulados dentro de componentes separados, para que o sistema fique mais fácil de manter.
- O sistema de software deve possuir um mecanismo uniforme que se aplique a todos os tipos de alterações (estruturais e comportamentais), para que o sistema seja fácil de entender e usar.
- O sistema deve permitir alterações de qualquer escala e até mesmo em seus aspectos fundamentais como o mecanismo de comunicação entre componentes, por exemplo.

2.5.2 Solução

Neste padrão, uma aplicação é dividida em 2 partes: uma que provê informação sobre propriedades selecionadas do próprio sistema chamada **meta-nível** e outra que implementa a lógica da aplicação chamada **nível base**. A informação do meta-nível mantém uma conexão de causa com o comportamento do sistema.

Diz-se que um sistema está conectado de forma causal a seu domínio, se qualquer alteração em um for refletida no outro, que se altera proporcionalmente para manter a conexão causal [Mae87]. Um sistema que controla a temperatura de um ambiente, por exemplo, possui dados que representam a temperatura do ambiente, esses dados estão conectados de forma causal à essa temperatura. Se a temperatura se eleva, os dados são alterados e o sistema dispara funções que acionam os aparelhos para resfriar o ambiente novamente, refletindo na temperatura. Dizemos que existe uma conexão de causa entre o sistema e a temperatura do ambiente, uma alteração em um é refletida no outro e vice e versa. Nos sistemas reflexivos, existe uma conexão de causa do sistema com ele próprio. Há uma conexão de causa entre o sistema (nível base) e sua auto-representação (meta-nível).

O nível base modela e implementa a funcionalidade ou lógica da aplicação. Em geral, no nível base de uma aplicação, existem componentes que colaboram entre si para oferecerem ao usuário as funcionalidades do sistema. O meta-nível consiste de um conjunto de meta-objetos. Cada meta-objeto encapsula informação selecionada acerca de um único aspecto da estrutura, comportamento, ou estado do nível base. No paradigma de objetos, isso significa que a representação de classes, métodos, atributos e objetos é materializada por metaobjetos. Esses meta-objetos tornam informações, que são implicitamente disponíveis, explicitamente acessíveis e modificáveis. A interface de um meta-objeto permite ao nível base acessar as informações que ele mantém e os serviços que ele oferece. Alterações no estado interno dos meta-objetos só são possíveis através do protocolo de meta-objetos ou por suas próprias computações. A maneira como os componentes do nível base estão relacionados com seus respectivos metaobjetos depende do modelo de reflexão da arquitetura reflexiva utilizada para implementar a aplicação. Existem 3 modelos de reflexão em linguagens orientadas a objetos baseadas em classes [Fer89]: modelo de meta-classe, de meta-objetos e de meta-comunicação. As Subseções seguintes descrevem esses modelos separadamente.

2.5.2.1 Modelo de meta-classe

É baseado em uma equivalência do meta-objeto e a classe de um objeto, veja Figura 2.7. O recebimento de uma mensagem *M* por um objeto *O*, instância de uma classe *C*, é equivalente ao recebimento da mensagem *HandleMsg* pela classe *C*. O método *HandleMsg*

é definido na meta-classe de *C* (*C_Class*) . Quando um objeto recebe uma mensagem, ele decide se usa o interpretador padrão ou se a mensagem é transferida para o meta-nível.

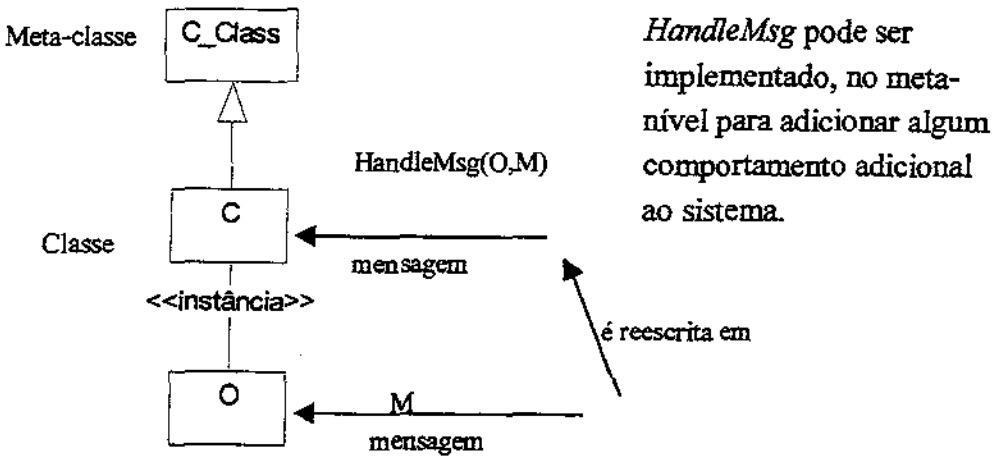


Figura 2.10: O modelo de reflexão de meta-classes

Conseqüências desse modelo:

- Cada instância da classe compartilha o mesmo interpretador de mensagem. Não há possibilidade de particularizar o interpretador de um único objeto. A modificação do interpretador pode ser feita substituindo a meta-classe, porém isso pode levar a inconsistências no sistema.
- Não é possível obter informações sobre o comportamento computacional dos objetos, como por exemplo o histórico de mensagens recebidas por eles.

2.5.2.2 Modelo de Meta-objeto

Neste modelo, os meta-objetos são instâncias de uma classe *Meta-Object*, ou de uma de suas subclasses. Cada objeto tem seu próprio meta-objeto, observe a Figura 2.8. Por exemplo, para cada objeto *O* pode existir um metaobjeto *M_O* que representa os aspectos estruturais e comportamentais de *O*. Cada mensagem *M* , enviada para um objeto *O*, é interceptada e dirigida ao seu metaobjeto *M_O*. O metaobjeto *M_O* encarrega-se da execução da mensagem *M*, que é realizada no nível base e controlada pelo meta-nível. O objeto emissor da mensagem *M* não toma conhecimento da computação reflexiva: ele envia a mensagem solicitando serviços de um objeto e recebe o resultado esperado, sem saber que a mensagem foi desviada para um metaobjeto. Esse modelo permite que seja possível realizar reflexão estrutural e comportamental. Reflexão estrutural visa modificar a

estrutura dos objetos e reflexão comportamental visa modificar o comportamento dos objetos.

Conseqüências:

- O meta-objeto de um objeto pode ser facilmente modificado.
- Um objeto pode ser monitorado individualmente.
- É possível definir novas formas de manipular mensagens criando subclasses de Meta-Object.

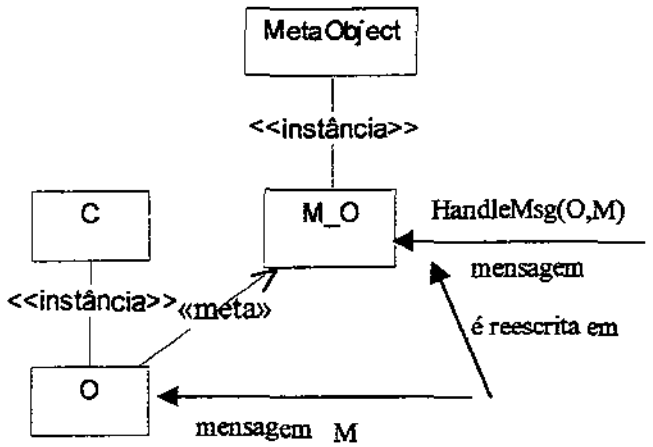


Figura 2.11: Modelo de reflexão de meta-objetos.

2.5.2.3 Modelo de meta-comunicações

Este modelo é baseado na materialização de mensagens. Uma mensagem *M* é materializada em um objeto (representado pelo círculo), veja a Figura 2.9, que pode reagir à mensagem *Send* enviada a ele sempre que uma mensagem *M* for enviada ao objeto *O*. O sistema está sempre no modo de reflexão, em outras palavras, pode-se dizer que o sistema está sempre "ciente" das mensagens enviadas, podendo fazer meta-computações sempre que for necessário.

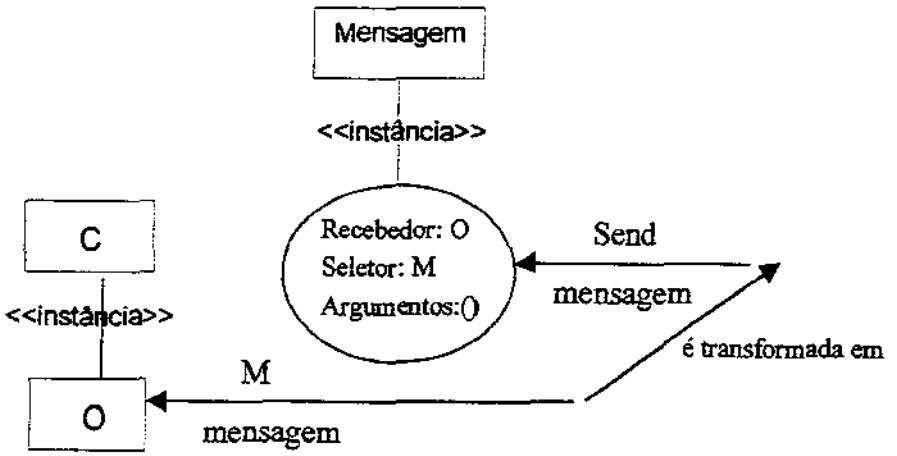


Figura 2.12: Modelo de reflexão de meta-comunicação.

Conseqüências deste modelo:

- É fácil fazer diferenciação das mensagens através da definição de subclasses da classe padrão Message, chamadas subclasses de comunicação.
- O uso de diferentes tipos de mensagens permite facilmente fazer extensões ao sistema, como a incorporação de troca de mensagens concorrentes, por exemplo.
- A desvantagem desse modelo é o fato de não ser possível a monitoração de objetos, somente de mensagens. Para aumentar o poder das meta-computações permitindo também a monitoração de objetos, este modelo pode ser utilizado juntamente com o modelo de meta-objetos.

2.5.3 Usos Conhecidos

Uma linguagem de programação possui uma arquitetura reflexiva, se ela reconhece a reflexão como um conceito fundamental de programação, provendo o uso de mecanismos de reflexão [LTM93]. Isso implica que a linguagem fornece construtores sintáticos para especificar codificações reflexivas que são entendidas e executadas pelo interpretador da linguagem, no caso de linguagens interpretadas, ou pelo pré-processador, no caso de linguagens compiladas. O interpretador ou o pré-processador garante a manutenção da conexão de causa entre os dados e os aspectos do sistema que eles representam. Desta maneira, os sistemas computacionais resultantes permitem o acesso às representações conectadas de forma causal durante a computação, e efetuação de modificações sobre essas representações a fim de alterar o comportamento do sistema.

Reflexão computacional tem sido aplicada em vários domínios. Maes [Mae87] propôs uma extensão reflexiva da linguagem KRS, chamada 3-KRS, cuja arquitetura reflexiva

segue o modelo de meta-objetos. A linguagem ABCL/R2 [MWY91] aplica a técnica de meta-nível à computação paralela. CLOS MOP [Pae88] foi a primeira tentativa de aplicar reflexão computacional a uma linguagem mais difundida entre programadores. OpenC++ [Chi93,Chi96] leva os conceitos de reflexão computacional para uma linguagem baseada em compilador amplamente utilizada, o C++. Smalltalk-80 [Gold83] facilita a implementação de facilidades reflexivas devido ao conceito de meta-classes presente na linguagem. Esta Seção descreve algumas dessas linguagens.

Reflexão computacional tem sido também utilizada para desenvolver sistemas computacionais tais como sistema operacional e sistema de janelas. Por exemplo, o Apertos [Yok92] é um sistema operacional completamente baseado na técnica e o Silica [Rao91] é um sistema de janelas baseado em CLOS em que o programador pode, por exemplo, alterar a forma como as imagens são desenhadas na janela por meio de reflexão computacional.

2.5.3.1 CLOS MOP- Protocolo de Meta-Objetos de Common Lisp Object System

CLOS [Pae88] é o Common Lisp padrão acrescentado de um modelo de objetos. A linguagem possui cinco elementos principais:

- Classes
- Slots
- Funções genéricas
- Métodos
- Combinações de métodos.

Classes contêm a descrição da estrutura e comportamento das instâncias. Slots armazenam dados das instâncias, são equivalentes a atributos em outras linguagem. Funções genéricas são o meio através do qual é possível enviar uma mensagem a um objeto. Ao chamar uma função genérica, passa-se como parâmetro a classe que implementa o apropriado corpo de código do método. Essa forma de seleção de métodos é chamada *class-sensitive*. A combinação de métodos permite ao programador especificar regras para execução de métodos. Algumas regras são pré definidas, outras são definidas pelo programador. Exemplos de regras pré definidas são as primitivas: **:before**, **:after** e **:around**. Essas primitivas permitem que algumas partes do código rodem antes, depois e ambos (antes e depois do método principal). Essas regras permitem aos programadores CLOS estruturarem a execução de métodos de várias formas de acordo com suas necessidades.

2.5.3.1.1 Arquitetura do Meta-nível

Os blocos de construção da linguagem são implementados por uma coleção de classes chamadas *classes metaobjeto*. A classe metaobjeto que descreve as classes são também chamadas meta-classes. Instâncias de classes metaobjetos são chamadas metaobjetos e os detalhes de comunicação entre estes são os chamados protocolos. O protocolo de CLOS possui:

1. Hierarquias de classes metaobjeto públicas cujas instâncias implementam os cinco blocos de construção descritos na linguagem.
2. Um protocolo que descreve quando metaobjetos são criados, que mensagens devem ser passadas entre eles e quais parâmetros devem ser usados. Um programador de sistemas pode estender ou modificar CLOS através de subclasses das classes.

A linguagem utilizada para implementar o MOP (*Metaobject Protocol*) é a própria CLOS. Essa abordagem de utilizar o produto da implementação de um sistema para construir o próprio produto é chamada **Meta-circular**.

Uma verdadeira arquitetura reflexiva é obtida através da adoção de interpretadores metacirculares. Eles permitem o acesso e alteração de seus interpretadores em tempo de execução. O programa é capaz de interromper sua computação, modificar seu interpretador e continuar com um processo modificado de interpretação. Esse mecanismo permite boas possibilidades de meta-computação, entretanto, ele tem baixa performance porque somam-se os *overheads* de interpretação da linguagem e do protocolo de metaobjetos.

2.5.3.2 Smalltalk

Smalltalk [Gold83] é um ambiente de programação cuja hierarquia de classes descende de uma única raiz - a classe *Object*. Smalltalk considera todas as suas entidades (classes, métodos, atributos, etc) como objetos.

2.5.3.2.1 Arquitetura do Meta-nível

A arquitetura de meta-nível é construída a partir de classes que descrevem suas entidades e suas hierarquias associadas. Essas classes podem ser visualizadas na Figura 2.10. Desta forma, Smalltalk apresenta um ambiente rico para a implementação de reflexão, mesmo não utilizando um interpretador meta-circular. Objetivando uma melhor divisão entre a computação de objetos e a computação reflexiva, Smalltalk provê o uso de meta-classes. A estrutura e interface de uma classe são especificadas pela sua meta-classe. As meta-classes controlam os mecanismos de instanciação, herança e compilação das classes. Elas fazem uso de métodos, classes e objetos que proporcionam facilidades reflexivas ao ambiente. Isso faz com que este modelo seja bastante dinâmico. O modelo de reflexão implementado

em Smalltalk é o modelo de metaclasses, pois as modificações são efetuadas nas classes, refletindo-se em todas as instâncias.

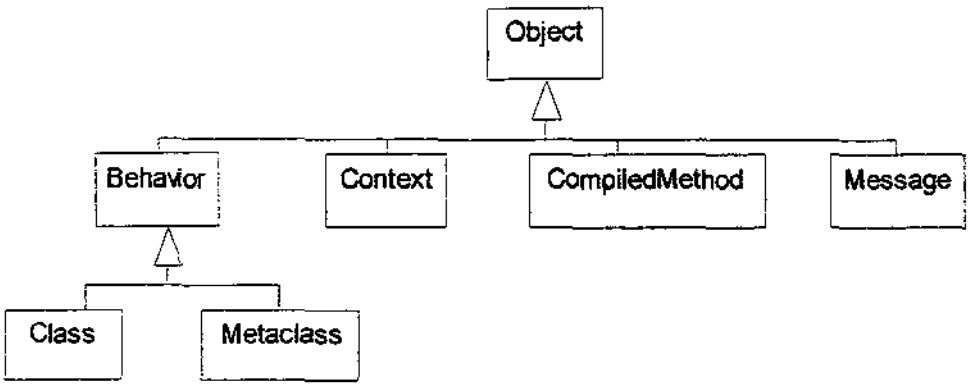


Figura 2.13: Hierarquia de classes parcial de Smalltalk-80

2.5.3.3 OpenC++ Versão 1.2

OpenC++ [Chi93] é também uma extensão reflexiva da linguagem C++. Ela permite que sejam definidos métodos e variáveis reflexivos. Isso significa que a implementação das chamadas a métodos e acesso a variáveis é extensível, ou seja, o programador pode acrescentar uma nova semântica, através da programação no meta-nível, para tais métodos e variáveis. Um objeto é controlado por um meta-objeto que altera o comportamento das chamadas a seus métodos e acesso às suas variáveis. O programador pode construir novos metaobjetos para adicionar novas funcionalidades, sem nenhuma alteração no compilador da linguagem. OpenC++ é uma extensão da linguagem C++, porque estende a semântica da linguagem de acordo com a necessidade do usuário.

2.5.3.3.1 Arquitetura do Meta-nível

OpenC++ adota o modelo de metaobjetos. Um objeto pode ser associado a um único meta-objeto e um meta-objeto não pode ser compartilhado por dois ou mais objetos. A arquitetura será descrita com o auxílio de um pequeno programa exemplo que mostra o comportamento de um objeto reflexivo e um objeto não reflexivo de uma mesma classe. Esta Seção mostra um exemplo de uma aplicação simples em OpenC++1.2. A Figura 2.11 mostra a implementação do nível base da aplicação e a Figura 2.12a mostra a implementação do meta-nível.

```
1.  Class Person {
2.  private:
3.     char * name;
4.     int age;
5.  public:
6.     Person(char * n, int a){name= n; age=a;}
7.     int Age(){return age;}
8.     char* Name(){return name;}
9.  //MOP reflect:
10.    int IncAge(){return ++age;}
11. };

12. //MOP reflect class Person : PrintMetaObj;

13. ---------
14. void main(){
15.    refl_Person student1("Billy", 14);
16.    printf("Nome: %s\n, student1.Name());
17.    printf("Idade: %d\n", student1.Age());
18.    printf("Idade: %d\n", student1.IncAge());
19.    Person student2("Mary", 24);
20.    printf("Idade: %d\n", student2.Age());
21.    printf("Idade: %d\n", student2.IncAge());
       }
```

Figura 2.14: Nível Base de uma aplicação

- Nível base: uma classe é associada a sua meta-classe por meio de diretivas em forma de comentários nos programas C++ (veja linha 12 da Figura 2.11). Variáveis e métodos são declarados como reflexivos também através de diretivas (linha 9 da Figura 2.11). Quando um objeto reflexivo é instanciado, é também criada uma instância de sua meta-classe, seu meta-objeto, que irá controlar o acesso às suas variáveis reflexivas e chamadas de seus métodos reflexivos. Uma mesma classe pode possuir objetos

reflexivos e não reflexivos. As linhas 15 e 19 da Figura 2.11 mostram, respectivamente, a criação de um objeto reflexivo e outro não reflexivo, ambos da classe *Person*.

- Meta-nível: as meta-classes, definidas pelo usuário, são subclasses da classe *MetaObj* (veja Figura 2.12). Esta classe implementa o protocolo de meta-objetos da linguagem. Ela define, entre outros, os seguintes métodos:
  - `void Meta_MethodCall(Id metthod, Id category, ArgPac & args, ArgPac &reply) -`
    este método é invocado quando um método reflexivo do nível base é chamado. É este método que permite que meta-computações sejam acrescentadas à execução de um método reflexivo.
  - `void Meta_HandleMethodCall(Id metthod, ArgPac & args, ArgPac &reply)-`
    este método é utilizado para executar um método reflexivo.

Os parâmetros principais são: *Id method*: identidade do método; *ArgPac & args* e *ArgPac & reply*: são os argumentos passados para o método e tipo retornado pelo método do nível base, que está sendo manipulado, respectivamente. O tipo ArgPac implementa uma pilha de tamanho variado de apontadores para elementos de quaisquer tipos, tratados como *void* *.

A Figura 2.12(a) mostra o código da aplicação, cujo nível base foi descrito na Figura 2.11, no meta-nível e a Figura 2.12(b) mostra o resultado do programa. A Figura 2.15 mostra o mecanismo de funcionamento das interrupções causadas pelo meta-nível na execução do programa. Observe, na Figura 2.12(b), o resultado da execução do programa. Note que para o objeto reflexivo, ouve a interceptação da mensagem *IncAge()* e a execução do método *Meta_MethodCall* da classe *PrintMetaObj*, no meta-nível, ilustrada na Figura 2.13, o que não ocorreu com o objeto não reflexivo.

O compilador OpenC++ 1.2 é um pré-processador que transforma o código OpenC++ (C++ com diretivas) em arquivos C++ e cria um arquivo de biblioteca chamado *reflect.a*, contendo o código gerado pelo pré-processador referente às diretivas. O compilador C++, em seguida, compila os programas C++ gerando arquivos objetos para serem linkados juntamente com *reflect.a* , a fim de gerar o código executável.

O *overhead* causado pela interceptação é anulado, pois as interceptações e outras meta-computações são amarradas em tempo de compilação.

```
Class PrintMetaObj: public MetaObj {
public:
        void Meta_MethodCall(Id m_id, Id category, ArgPac & args,
ArgPac & reply)
        {printf("*** %s foi chamado. \n", Meta_GetMethodName(m_id));
        Meta_HandleMethodCall(m_id, args, reply);}
}
```

(a)

```
Nome: Billy
Idade: 14
*** IncAge foi chamado.
                Idade:15
Nome: Mary
Idade: 24
Idade:25
```

(b)

Figura 2.15: (a) Meta-nível da aplicação (b) Resultado exibido pela aplicação.

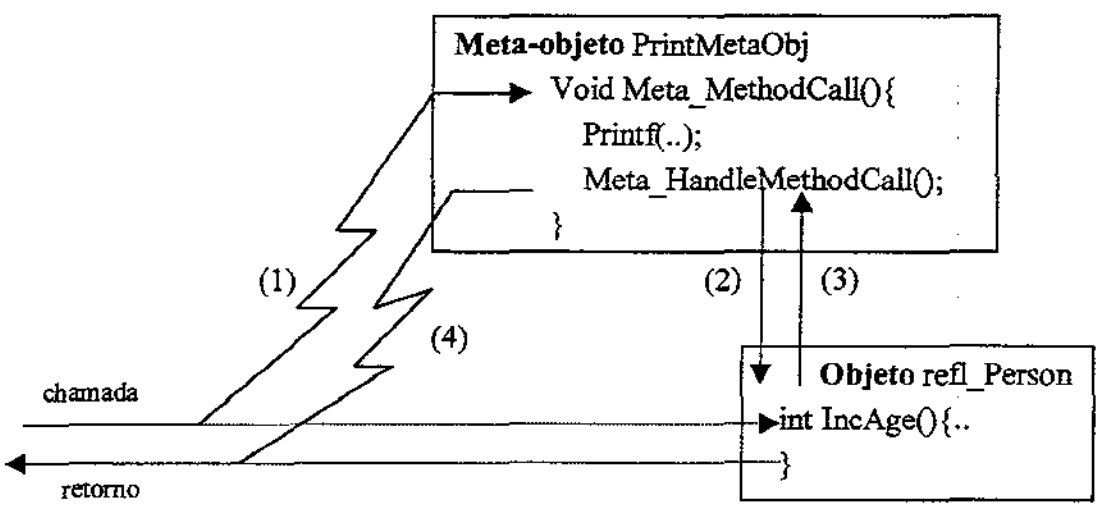


Figura 2.16: Execução de um método reflexivo em OpenC++1.2

2.5.3.4 OpenC++ Versão 2.0

O compilador OpenC++ 2.0 [Chi96] é uma extensão de C++ que permite o uso de reflexão computacional.

2.5.3.4.1 Arquitetura do Meta-nível

Os programas do nível base são representados por classes pré-definidas no protocolo, as quais podem ser estendidas por subclasses definidas pelo programador. As principais classes do protocolo são:

- **Ptree**: representa uma árvore de *parse* do programa, implementada por uma lista encadeada. Possui métodos sobrecarregáveis que permitem ao programador obter o significado sintático do texto do programa e modificá-lo.
- **Class**: representa as definições de classe (é uma meta-classe) e controla a tradução do programa fonte. Metaobjetos do tipo **Class** implementam o protocolo de meta-objetos.

OpenC++ 2.0 é meta-circular, isso implica que os programas do meta-nível são também programas OpenC++. Meta-classes podem também ter meta-classes. A meta-classe de **Class** é a classe pré-definida *Metaclass*. *Metaclass* é a meta-classe *default* das meta-classes do meta-nível. Metaclass é subclasse de Class e, portanto, sua meta-classe é Metaclass, ou seja, ela própria. Assim, é interrompida a meta-circularidade de OpenC++ 2.0.

O compilador consiste de três fases: pré-processamento, tradução de fonte OpenC++ para fonte C++ e compilação C++. No pré-processamento, o código OpenC++ é lido e materializado nos metaobjetos. Na fase de tradução, os metaobjetos são transformados em fragmentos de código já devidamente alterados para incorporar o comportamento reflexivo. Caso o código gerado ainda possua comandos OpenC++, o processo é repetido até que se obtenha apenas fragmentos contendo código C++ puro.

OpenC++ permite grandes possibilidades de meta-computações, pois oferece flexibilidade para o programador. Entretanto, seu protocolo é bastante complexo, difícil de ser utilizado.

A garantia de desempenho é obtida através da divisão do processo de compilação em duas fases (pré-processamento e tradução). O *overhead* causado pela interceptação é anulado, pois as interceptações o outras meta-computações são amarradas em tempo de compilação. O código obviamente é maior e mais lento que um código sem funcionalidades reflexivas, não pela forma que a arquitetura as incorpora, mas devido às próprias funcionalidades.

O protocolo de metaobjetos de OpenC++ 1.2 é mais simples para ser utilizado que o OpenC++ 2.0, porém um pouco mais limitado. OpenC++ 1.2 não permite, por exemplo, que dois objetos em uma mesma hierarquia possuam metaobjetos de classes distintas devido à forma que seu protocolo é implementado. OpenC++2.0 permite maior flexibilidade, pois disponibiliza todo o código do programa do nível base no meta-nível através do meta-objeto Ptree, no entanto, a modificação desse código através de Ptree não é trivial.

2.5.3.5 Protocolos de meta-objetos em Java

Os protocolos de meta-objetos em Java são mais recentes e ainda não estão consolidados, não permitindo determinar um único modelo de reflexão em Java. Nesta seção descrevemos sucintamente três desses protocolos: MetaJava [GOL97], Reflective Java[Wu97] e Guaraná [OGB98].

2.5.3.5.1 MetaJava

O protocolo MetaJava [GOL97] implementa reflexão estrutural e comportamental em Java através da extensão da máquina virtual Java – JVM. MetaJava permite associação de meta-objetos a objetos, classes e referências. O protocolo utiliza eventos síncronos para realizar a transferência da computação do nível base para o meta-nível. Quando o meta-objeto é associado a um objeto, todos os eventos originados no objeto são transmitidos ao meta-objeto correspondente. Se o meta-objeto é associado a uma classe, os eventos nas instâncias da classe são transmitidos ao meta-objeto; e se ele é associado a uma referência, somente as operações que usam esta referência são transmitidas ao meta-objeto. MetaJava modificou a JVM de forma a disponibilizar no meta-nível os dados da aplicação no nível base, provendo bastante flexibilidade ao programador do meta-nível. A reflexão estrutural viola o encapsulamento das classes e a meta-circularidade, uma de suas características, pode provocar conflitos semânticos.

2.5.3.5.2 Reflective Java

O protocolo Reflective Java [Wu97] oferece somente reflexão de métodos. Assim como OpenC++ , Reflective Java utiliza um pré-processador para implementar a reflexão. O pré-processador cria uma classe reflexiva como subclasse da classe do objeto referente do nível base. A classe reflexiva possui uma instância de uma classe *metaobject,* que implementa a inteceptação de chamadas aos métodos do referente. Os objetos podem ser associados a meta-objetos estatica ou dinamicamente.

2.5.3.5.3 Guaraná

A arquitetura reflexiva do Guaraná [OGB98], assim como MetaJava, foi implementada modificando a máquina virtual Java. O núcleo do Guaraná realiza os seguintes mecanismos básicos: (i) interceptação e materialização das operações; (ii) amarração dinâmica e invocação de objetos do meta-nível para tratar as operações interceptadas; e (iii) manutenção das meta-informações estruturais. Cada objeto pode ser diretamente associado a zero ou mais meta-objetos. Não é permitido a um objeto referenciar seus meta-objetos. O acoplamento entre objeto e meta-objeto é garantido pela interceptação e materialização de todas as operações enviadas ao objeto referente e mecanismos de amarração dinâmica. Guaraná permite também a associação indireta de múltiplos meta-objetos a um objeto, através de um meta-objeto especial chamado *Compose*, o qual é responsável por garantir políticas que definem a estrutura e ordem do fluxo das operações delegadas aos meta-objetos.

2.6 Resumo

A orientação a objetos tem se mostrado uma tendência promissora como paradigma para desenvolvimento de sistemas. As técnicas mais recentes de construção e documentação de software também se baseiam no paradigma. Padrões de software, descrito na Seção 2.2, é um tópico de pesquisa amplamente estudado atualmente, pois visa ampliar o nível de reutilização proporcionado pelo paradigma de objetos. Padrões de software não necessariamente está preso ao paradigma, mas seu uso é efetivo na construção de sistemas orientado a objetos. O uso de reflexão computacional (Seção 2.5) permite que serviços administrativos (ou requisitos não funcionais) como tolerância a falhas e distribuição, por exemplo, implementados no meta-nível, sejam fornecidos de forma transparente para a aplicação implementada no nível base. No que diz respeito ao desenvolvimento de interfaces homem-computador o uso das linguagens orientadas a objetos e conceitos como padrões de software, *frameworks* (descrito no próximo capítulo) e reflexão computacional podem facilitar grandemente o desenvolvimento de sistemas interativos mais fáceis de estender e manter.

Capítulo 3

Frameworks para Interfaces Homem-Computador

Este capítulo discute alguns *frameworks* para interfaces homem-computador (*GUI frameworks*). A Seção 3.1 descreve padrões de arquitetura (dentre eles o padrão MVC *Model/View/Controller*) utilizados freqüentemente na definição da arquitetura desses *frameworks*. Esta Seção discute o problema da falta de independência de diálogo nos sistemas interativos e a forma como o MVC busca solucionar o problema. Conforme visto no Capítulo 1, o forte acoplamento entre aplicação e interface gráfica em um sistema interativo foi o problema que motivou o desenvolvimento deste trabalho. As Seções seguintes visam mostrar os trabalhos atuais que objetivam facilitar a construção de sistemas interativos. A Seção 3.2 conceitua *frameworks* orientados a objetos e situa este trabalho no domínio dos frameworks para desenvolvimento de aplicações com interfaces gráficas. A Seção 3.3 descreve a estrutura e comportamento de alguns *frameworks* para interfaces homem-computador mais conhecidos. Finalmente, a Seção 3.4 faz um resumo deste capítulo.

3.1 Padrões de Arquitetura para Aplicações Interativas

Esta Seção descreve alguns padrões de arquitetura conhecidos para construção de aplicações interativas. Esses padrões são freqüentemente utilizados na construção de *frameworks* para interfaces homem-computador. Esta Seção faz também uma comparação entre os padrões descritos.

3.1.1 Model-View-Controller

Model-View-Controller (MVC) foi idealizado por Reenskaug e primeiramente implementado no ambiente de programação de Smalltalk-80 [Gold83]. O MVC provê a

mais conhecida e utilizada organização arquitetural para sistemas interativos. Para citar alguns exemplos: o padrão aparece na arquitetura do MacApp [Alger90] (um *framework* para a construção de aplicações Macintosh), uma variação do padrão é utilizada no ET++ [WGM89] (*framework* para aplicações Unix) e MFC - Microsoft Foundation Class - (*framework* para aplicações Windows) segue princípios do padrão. Ele foi criado visando proporcionar separação de serviços aos sistemas interativos. Esta Seção descreve o padrão segundo os elementos: problema, solução, usos conhecidos e conseqüências.

3.1.1.1 Problema

Uma aplicação interativa é composta de um núcleo funcional (ou aplicação) e da interface do usuário (*GUI- Grafical User Interface*) [Hhi89], conforme Figura 3.1.

Figura 3.1 : Divisão lógica de um sistema Interativo

Interface com o usuário está sempre sujeita a alterações. Quando se tem um sistema fortemente acoplado a sua interface gráfica, uma alteração na apresentação do sistema torna-se difícil e mais sujeita a erros. Alterações na interface com o usuário são bastante freqüentes em um sistema. Quando uma funcionalidade nova é introduzida na aplicação, alterações devem ser feitas na interface gráfica para que o usuário possa utilizar essas novas facilidades. Diferentes usuários, muitas vezes, preferem utilizar as funcionalidades da aplicação de formas diferentes. Um usuário pode querer trabalhar com teclado enquanto outro prefere utilizar o mouse. Consequentemente, é desejável que tenha facilidade em acrescentar suporte para as várias formas de interação. Ou ainda, um cliente pode querer adaptar uma interface de um sistema para ser utilizada em outras plataformas com uma apresentação padrão diferente. Assim, é desejável que a arquitetura do sistema seja flexível a ponto de permitir que tais alterações sejam feitas da maneira mais fácil possível. Porém, é necessário que algumas forças sejam garantidas:

- A mesma informação é apresentada de diferentes formas. O sistema deve permitir que um dado da aplicação possua diferentes apresentações.
- Deve haver consistência entre os dados da aplicação e suas apresentações. A apresentação e comportamento do sistema devem refletir as manipulações de dados

imediatamente. Existe uma conexão de causa entre a aplicação e sua apresentação (interface gráfica) que deve ser mantida em um sistema interativo.

- A interface deve dar suporte a diferentes *look and feel* padrões ou o porte da interface para uma outra plataforma não deve afetar a aplicação.
- A alteração da interface deve ser facilitada e possível até mesmo em tempo de execução.

3.1.1.2 Solução

O MVC divide uma aplicação interativa entre três componentes como mostra a Figura 3.2, Modelo, Visão e Controlador que representam o processamento, a representação dos dados (saída), e manipulação de entrada dos dados, respectivamente.

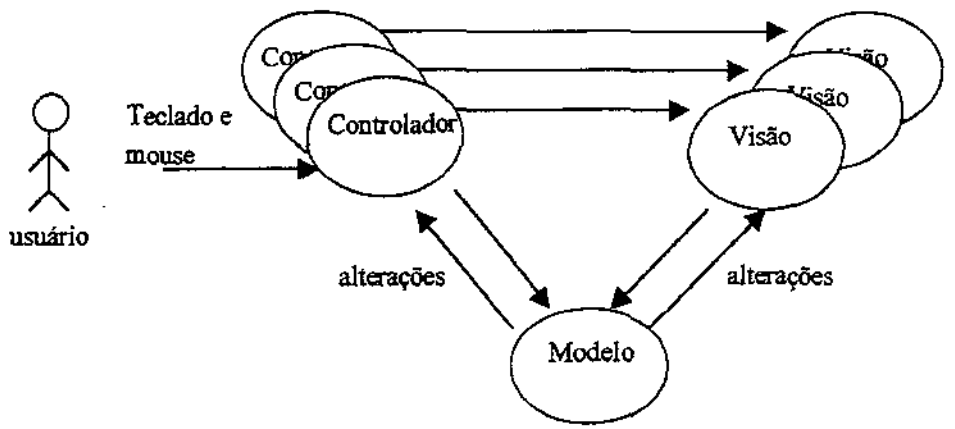


Figura 3.2: A interação no padrão MVC - um modelo com múltiplas visões.

Modelo

O componente Modelo encapsula os dados específicos do domínio e a funcionalidade da aplicação. Pode-se citar como exemplo os inteiros (representando contadores), arranjos de caracteres (em editores de textos simples e ainda todas as formas de implementação de estruturas de dados como listas encadeadas, pilhas, registros, etc. O modelo é independente de representações específicas de saída (Visão) e de qualquer comportamento de entrada (Controlador).

Visão

O componente Visão encapsula os objetos gráficos pertencentes à interface. Ele controla a apresentação visual dos objetos do modelo. Por exemplo: genericamente, a visão é responsável por atualizar ou rolar uma janela ou, especificamente, ela é responsável por mostrar dados representados por um gráfico de pizza, por exemplo. Podem existir múltiplas

Visões de um mesmo Modelo como mostra a Figura 3.2. Cada Visão possui um Controlador associado, para controlar as interações com o usuário. A Visão deve conhecer o modelo o qual está representando, mas não precisa conhecer as outras Visões.

Controlador

Controladores recebem eventos de entrada do usuário como movimentação de mouse, ativação de botões do mouse e entrada via teclado. Esses eventos são traduzidos para requisições de serviços aos componentes Modelo e Visão. O componente Controlador controla a interação com o usuário, é através dele que o usuário interage com o sistema. O Controlador deve conhecer a Visão e Modelo com os quais ele está associado.

Esta arquitetura permite que um Modelo esteja associado a múltiplos pares Visão-Controlador. Isso requer que um mecanismo de propagação de alteração seja implementado, pois se o Modelo é alterado a partir do Controlador de alguma Visão, é necessário que as outras Visões sejam notificadas, a fim de manter a consistência do sistema. Esse mecanismo de propagação de alteração[4] é implementado pelo padrão de projeto Observador.

3.1.1.2.1 Estrutura Estática do Padrão

A Figura 3.3 mostra o projeto do padrão de arquitetura MVC utilizando o padrão de projeto Observador na implementação do mecanismo de propagação de alteração entre os componentes. O padrão de projeto Observador freqüentemente é utilizado no projeto do MVC, pois ele implementa de forma elegante a dependência dos componentes Controlador e Visão em relação ao Modelo. Observe que Visão e Controlador observam Modelo e Modelo está associado a zero ou mais observadores que são Visões e Controladores. Um *framework* para desenvolvimento de interfaces homem-computador poderia ser construído a partir da estrutura da Figura 3.3.

Classe Model:

A classe modelo encapsula dados e exporta funções para acesso a esses dados e processamento dos mesmos. O usuário acessa às funções de processamento dos dados através do Controlador e a visão utiliza as funções de acesso aos dados para manter atualizada a apresentação dos mesmos. Em suma, as principais responsabilidades do Modelo são:

- Implementar a funcionalidade da aplicação;

[4] Do inglês *change propagation.*

- Registrar as Visões e Controladores dependentes;
- Notificar os dependentes de suas alterações.

Classe View:

Diferentes visões apresentam a informação ao usuário de diferentes formas. O mecanismo de propagação de alteração, disparado pelo Modelo, irá chamar o método *update* para as visões dependentes. Cada Visão implementará seu específico método *update*. Quando *update* é chamado, a visão deve recuperar os dados do modelo para se atualizar. Pode-se listar as seguintes responsabilidades para a Visão:

- Criar e iniciar seus Controladores associados;
- Mostrar a informação ao usuário;
- Implementar o método *update;*
- Recuperar dados do Modelo.

Classe Controller:

O componente Controlador recebe as entradas do usuário como eventos, interpreta esses eventos para transformá-los em requisições a serviços da interface do Modelo e da Visão. Pode-se citar como responsabilidades do Controlador:

- Reconhecer entradas do usuário como eventos;
- Traduzir eventos para requisição de serviços (handleEvent);
- Implementar o procedimento *update* - muitas vezes o controlador é alterado, quando determinado dado do modelo é alterado, alguns serviços podem se tornar ou deixar de estar disponíveis de acordo com o estado do Modelo.

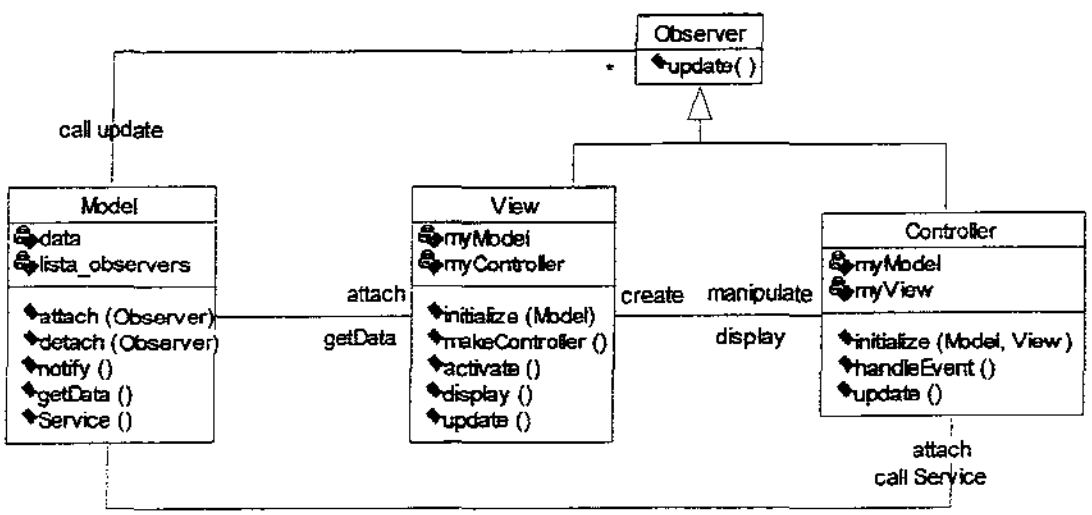


Figura 3.3 : Projeto do MVC utilizando o padrão Observador.

Observe que a estrutura mostrada na Figura 3.3 (padrão Observador) permite a implementação das responsabilidades listadas para cada componente.

3.1.1.2.2 Comportamento Dinâmico

O comportamento dinâmico do padrão é mostrado através de diagramas de colaboração. O diagrama da Figura 3.4 mostra o processo de criação dos objetos[5] do sistema. Inicialmente, uma instância do Modelo[6] é criada, em seguida é criada uma instância da Visão (mensagens 1 e 2). A Visão é então iniciada com uma referência para o Modelo (mensagem 3). A Visão se inclui na lista de observadores do Modelo (mensagem 4). Em seguida, ela cria o seu Controlador, inicia o mesmo com as referências para o Modelo e para si própria (mensagens 5, 6 e 7). Finalmente, o Controlador também se inclui na lista de observadores do Modelo.

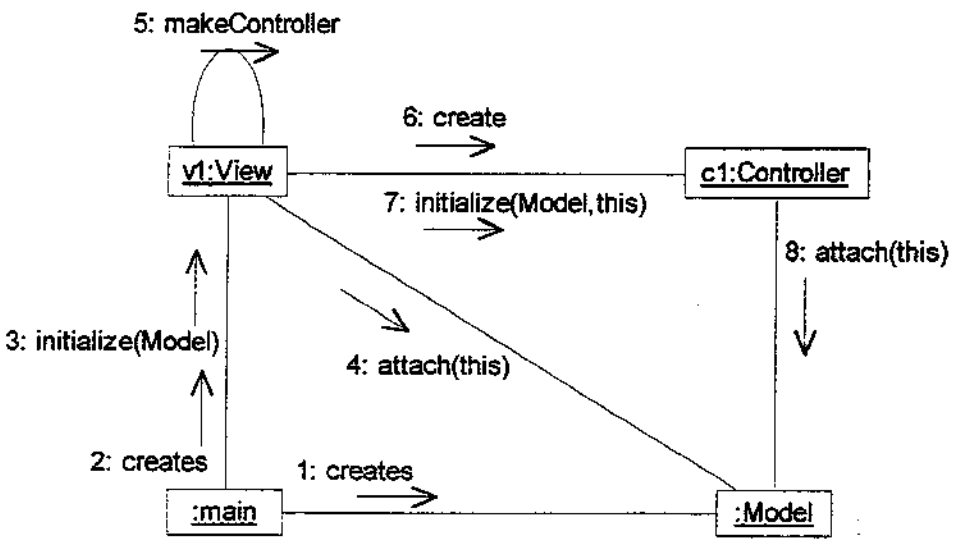


Figura 3.4 : Iniciação do sistema interativo.

A Figura 3.5 mostra uma seqüência de execução em que uma alteração do Modelo dispara o mecanismo de propagação de alteração. Suponha que o acionamento de um botão do *mouse* tenha causado a execução do método *handleEvent* do Controlador (mensagem 1). O

[5] O objeto *Main* representado na Figura serve apenas para representar o início da execução do sistema.

[6] O objeto Modelo no diagrama não possui identificação. Isso acontece porque existe apenas um objeto Modelo no sistema, os objetos Visão e Controlador estão identificados porque podem existir outros, além dos representados na fugura.

Controlador requisita então a execução dos serviços referentes ao acionamento do mouse (mensagem 2). Suponha que a execução de tal serviço tenha causado alteração nos dados do Modelo. O Modelo, na execução do método *Notify*, chama *update* para seus dependentes (mensagens 3 e 4). A Visão, na execução de *update*, recupera dados do Modelo e os mostra para o usuário (mensagens 5 e 6), semelhantemente, o Controlador também recupera dados do modelo para a habilitação ou desabilitação alguma função (mensagem 8).

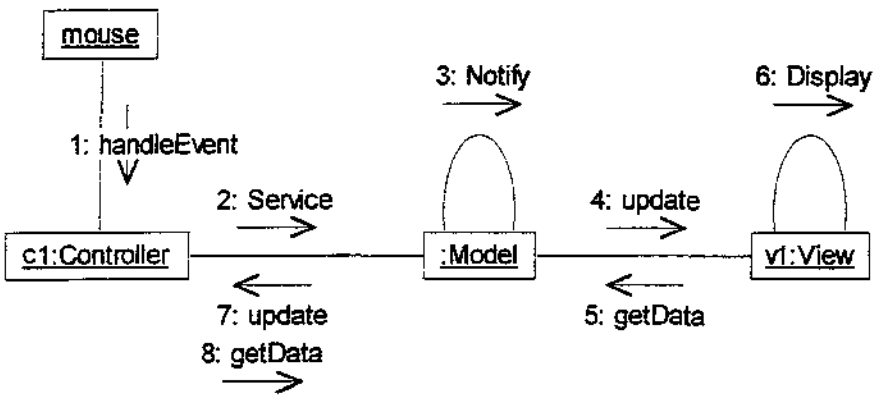


Figura 3.5 : Funcionamento do mecanismo de propagação de alteração.

3.1.1.3 Usos Conhecidos:

O *framework* para interface com o usuário do ambiente de **Smalltalk-80** [Gold83] foi a primeira implementação do padrão MVC. O padrão permitiu a criação de componentes reutilizáveis que são compartilhados pelas ferramentas que fazem parte do ambiente de desenvolvimento Smalltalk.

MFC (Microsoft Foundation Class) - utiliza também uma variante do MVC que une em um só componente a Visão e o Controlador. Os eventos de uma interface estão, na verdade fortemente acoplados a ela. Esta abordagem torna-se mais eficiente, porém sacrifica a facilidade de alteração do controlador. Essa variante é chamada de *Document-View* [BMR96], ou simplesmente MV (*Model-View*).

MacApp, um *framework* para desenvolvimento de aplicações Macintosh, usa uma variação do padrão para implementar as alterações da visões. Existe um mecanismo de propagação de alteração do MVC baseado em uma política de invalidação de regiões de uma visão afetada pela alteração do modelo, implementada pela *toolbox* gráfica do Macintosh. A Seção 3.3.1 descreve mais detalhadamente o MacApp.

ET++ [WGM88]- O *framework* de aplicação ET++ também usa uma variação do MVC. No ET++ , o controlador se separa entre os componentes Visão e Modelo. Grande parte da funcionalidade do Controlador fica acoplada à Visão e uma pequena parte ao Modelo.

3.1.1.4 Conseqüências

Dentre as vantagens da utilização do MVC pode-se citar:

- Múltiplas visões do mesmo modelo;
- Alteração de Visões e Controladores com pouco esforço.
- *Framework* potencial - é possível basear um *framework* de aplicação neste padrão. Os exemplos citados no item "usos conhecidos" comprovam esta afirmação.

Entretanto, o padrão de arquitetura MVC possui algumas limitações, dentre as quais pode-se citar:

- Aumento da complexidade na separação de Visões e Controladores [WGM89]. Os autores do *framework* ET++, afirmam que separar modelo e visão de estruturas simples como de um item de menu, por exemplo, aumenta a complexidade, reduz a eficiência e não se ganha flexibilidade.
- O mecanismo de propagação de alteração causa uma alteração em todas as visões para uma simples ação do usuário. Muitas vezes, uma visão não necessita ser notificada de qualquer alteração, porque está minimizada, por exemplo.
- Existem dependências entre Visões e Controladores que dificultam a reutilização de um ou outro separadamente.
- Existem dependências entre Visões e Controladores em relação ao modelo. Viu-se no modelo de objetos da Figura 3.3 que a Visão e Controlador devem conhecer o Modelo. As dependências entre esses módulos contradizem o conceito de independência de diálogo introduzido por Hartson e Hix [HHi89], que diz que o software da aplicação deve ser desenvolvido independentemente do software da interface. Essas dependências dificultam e, em muitos casos, impedem que a aplicação ou interface sejam reutilizadas independentemente. A adição de uma interface gráfica a uma aplicação já implementada, por exemplo, torna-se uma tarefa extremamente trabalhosa, pois muitas alterações deverão ser feitas nos códigos fonte da aplicação [CRB97].
- Ineficiência no acesso aos dados do Modelo na Visão.
- Dificuldade de utilização do MVC com ferramentas modernas de interface do usuário. As ferramentas, em geral, geram códigos que não seguem a modularização imposta pelo MVC.

3.1.1.5 Evolução do MVC

O MVC é tido como o padrão mais apropriado para a implementação de interações quando se trata da definição de componentes colaborativos de um *frameworks* de aplicações [Lew95]. Os mais bem construídos *frameworks* de aplicações interativas foram baseados no MVC ou em uma de suas variantes.

Criado por Reenskaugh e primeiramente utilizado na íntegra em Smalltalk-80 [Gold83], o MVC é o modelo mais antigo e mais citado como padrão de arquitetura para *frameworks* para interfaces homem-computador. Os *frameworks* mais recentes utilizam variações do MVC. As alterações, em geral, são feitas no componente Controlador. Há uma certa tendência em unir os componentes Controlador e Visão, criando assim o padrão MV (*Model-View*). O Andrew Toolkit [Pal88], MFC e o ICpak 201 [Kno89] são exemplos de implementações do MVC com essa característica. O MacApp e o ET++ dividem o componente Controlador entre os componentes Modelo e Visão através de uma classe comum entre esses dois componentes, responsável pelo tratamento de eventos. Os eventos são recebidos pela Visão e passados para uma cadeia de tratadores de eventos (objetos do modelo ou visão) que decidem tratar ou repassar o evento para o próximo objeto da cadeia. Nesses frameworks, o conceito de visões dependentes de um modelo é estendido para dependência de objetos em geral. O mesmo mecanismo de propagação de alteração utilizado para visão e modelo, definido pelo MVC, é utilizado por quaisquer objetos. Os objetos da Visão, Modelo ou Controlador podem ser observadores ou observados. Isso não prejudica a modularidade do *framework*, até mesmo proporciona uma certa homogeneidade aos objetos, que possuem uma funcionalidade em comum: a capacidade de implementar a notificação de suas alterações a todos os seus dependentes.

A razão dessa tendência em unir os componentes Controlador e Visão é explicada pelo ganho de eficiência que se adquire, quando o comportamento dinâmico da interface do usuário é acoplada à sua estrutura estática. Por outro lado, perde-se em modularidade e clareza. Utilizando essa abordagem, a separação de serviços entre Modelo e Visão deve ser bem clara, pois a tendência em unir os componentes passa a ser maior, visto que o que dita o comportamento dinâmico de uma interface é a lógica da aplicação. O MVC++ [Jaa91] é uma variação do MVC que visa usufruir dos benefícios do modelo MV sem perder a modularidade e clareza proporcionados pelo MVC. A próxima Seção descreve o padrão de arquitetura MVC++.

3.1.2 MVC++ - Model View Controller ++

O MVC++ [Jaa95] é um padrão de arquitetura idealizado como uma variação do padrão de arquitetura original MVC (*Model-View-Controller*) no projeto de softwares interativos. O problema que o MVC++ procura solucionar é o mesmo do MVC, descrito na Seção 3.1.1.1,

a solução é um pouco distinta e as conseqüências de uso também. As Subseções seguintes descrevem a solução e as conseqüências da utilização do padrão.

3.1.2.1 Solução

O MVC++ define três componentes em um sistema interativo: Modelo, Visão e Controlador, como mostra a Figura 3.6. Modelo, assim como no original MVC, representa a lógica da aplicação. Visão representa os objetos gráficos e mais os receptores das entradas do usuário, que no MVC original eram implementadas pelo Controlador. O Controlador deste padrão é responsável apenas pelo gerenciamento dos objetos do Modelo. A Figura 3.6 mostra a estrutura básica do padrão e o papel de cada componente em uma interação com um usuário.

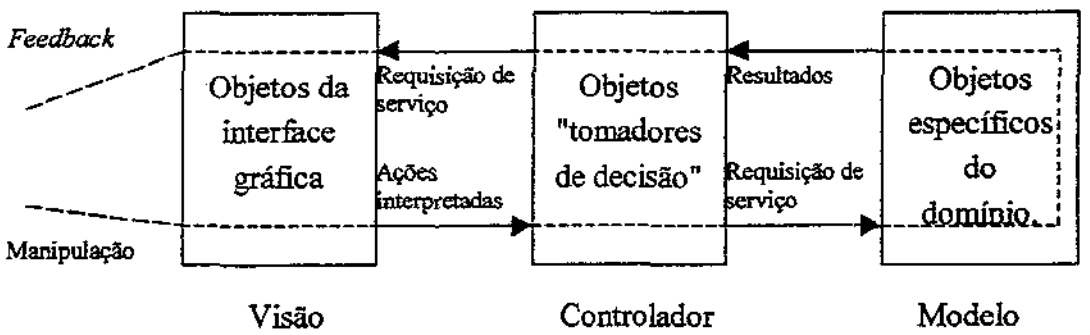


Figura 3.6 : Estrutura e comportamento do MVC++

Os objetos da Visão formam a camada do sistema que é visível ao usuário final. Eles apresentam os objetos do domínio para o usuário, recebem ações do usuário, mas não decidem como irão tratar essas ações. Os objetos da Visão passam as ações do usuário interpretadas para os objetos do Controlador decidirem quais objetos do Modelo tratarão as requisições do usuário. O Controlador integra objetos do Modelo e Visão de acordo com a aplicação específica. Os componentes podem ser organizados estaticamente como mostra a Figura 3.7. Observe que a principal diferença entre o MVC++ e o MVC original é a função do Controlador. Nesta abordagem, o usuário não mais interage com o Controlador e sim com a própria Visão, e a função do Controlador passa a ser de integração entre os principais componentes: Modelo e Visão.

A implementação do MVC++ sugerida em [Jaa95] é a seguinte:

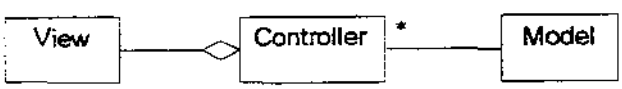


Figura 3.7: modelo de objetos básico do MVC++

O padrão permite que um Modelo esteja associado a múltiplas Visões. Para tanto, pode-se utilizar o mesmo mecanismo de propagação de alteração do MVC: o padrão de projeto Observador.

3.1.2.2 Conseqüências

As vantagens da utilização do MVC++ são as mesmas da utilização do MVC, listadas na Seção 3.1.1.4. O MVC++, no entanto, visa minimizar uma das desvantagens do MVC: a complexidade de manter serviços semelhantes em componentes distintos, como é o caso da recepção dos eventos do usuário desacoplados da interface propriamente dita. É mais fácil e mais eficiente unir estes serviços em um único componente. Alguns *GUI frameworks* construídos a partir de variações do MVC utilizaram essa abordagem (veja seção 3.1.1.5). A grande vantagem do MVC++ em relação às variações que surgiram do MVC é a existência de um componente entre Modelo e Visão que faz o serviço de integração entre os dois componentes. Isso permite que as visões sejam independentes do modelo que elas representam.

3.1.3 PAC – Presentation Abstraction Control

O *Presentation Abstraction Control* (PAC) [Cou87] é um padrão de arquitetura para construção de sistemas interativos que define uma estrutura na forma de uma hierarquia de agentes colaborativos. Neste contexto, um agente denota um componente que inclui tratadores e transmissores de eventos, elementos que mantêm e alteram os seus estados e podem produzir novos eventos. O contexto de aplicação do padrão é a construção de sistemas interativos com o uso de agentes. Agentes especializados em interação homem-computador aceitam eventos do usuário e apresentam os dados a eles.

Em arquiteturas de agentes colaborativos, cada agente é especializado em uma tarefa específica e todos os agentes juntos provêem a funcionalidade do sistema.

As principais forças que a solução deve garantir são:

- Agentes interativos provêem suas próprias interfaces, pois cada funcionalidade pode exigir um tipo distinto de interação.
- Agentes devem manter seus próprios dados e estado.
- Mudanças em um agente não devem afetar outros agentes.

3.1.3.1 Solução

O sistema interativo deve ser estruturado como uma hierarquia de agentes PAC. Um agente PAC consiste de três componentes: Apresentação, Abstração e Controle. O componente Apresentação implementa a apresentação de um agente e a interação com o usuário. O componente Abstração possui os dados de um agente e as operações que os agentes manipulam. O componente Controle conecta a Apresentação à Abstração e provê a funcionalidade que permite a comunicação com outros agentes PAC. A hierarquia de agentes PAC possui três tipos de agentes: um agente de alto nível, agentes intermediários e agentes de baixo nível. O agente de alto nível provê a funcionalidade básica do sistema e inclui aspectos da interface, que não podem ser atribuídos à subtarefas como janelas, barras de menu ou uma caixa de diálogo que mostra informações sobre a aplicação. O agente de baixo nível representa conceitos semânticos auto-contidos nos quais o usuário pode atuar, como por exemplo um formulário. Agentes intermediários representam combinações ou relacionamentos entre agentes de baixo nível.

Suponha um sistema que apresente dados da produção de uma fábrica em forma de um gráfico de pizza e de barras. O sistema seria divido em agentes PAC da seguinte forma:

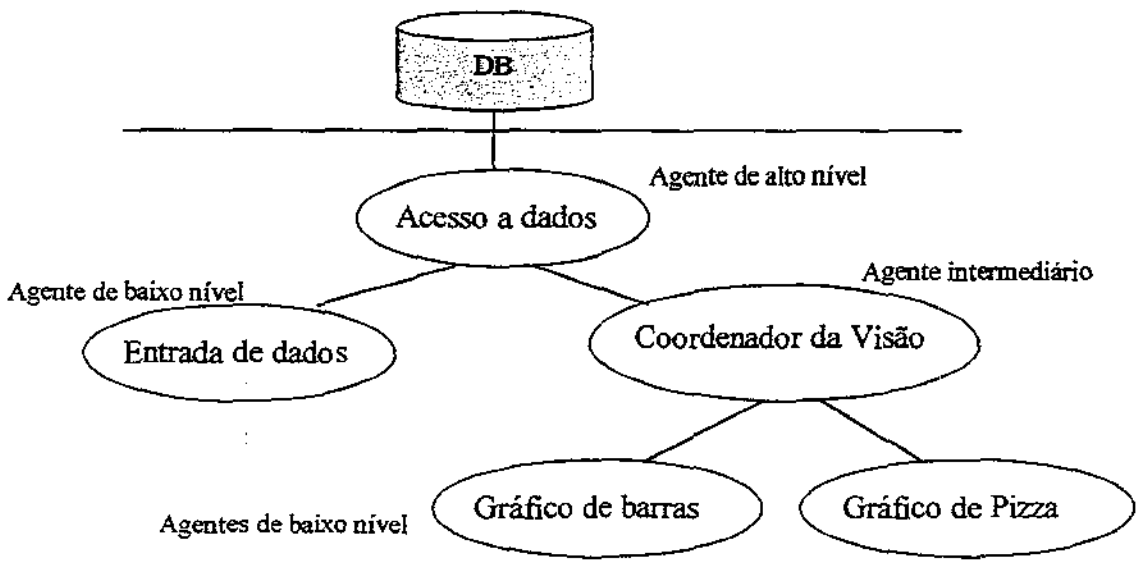
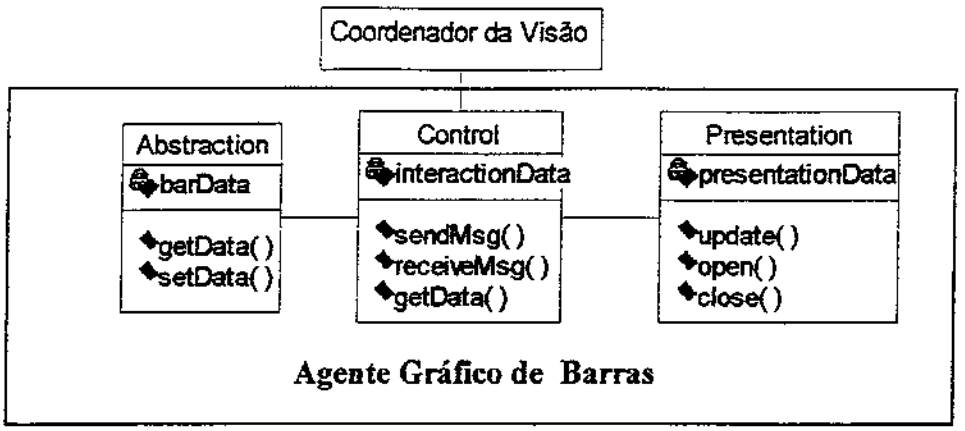


Figura 3.8 : Exemplo de uso de agentes PAC.

Cada um dos agentes possui Apresentação, Abstração e Controle. Veja o exemplo do agente PAC Gráfico de Barras:



O funcionamento dinâmico do sistema é gerenciado pelos componentes Controles dos agentes, principalmente do agente de alto nível e dos agentes intermediários. O mecanismo de propagação de alteração é descrito abaixo:

Suponha uma alteração dos dados do repositório através do agente Entrada de Dados. O agente de alto nível Acesso a Dados deve então avisar ao agente Coordenador da Visão das alterações. O Coordenador da Visão notifica os agentes de baixo nível que dependem dos dados: Gráfico de Barras e Gráfico de Pizza. Os agentes requisitam ao Coordenador da Visão os novos dados. O Coordenador da Visão solicita os novos dados ao agente Acesso a Dados. Os dados são então transmitidos do agente de alto nível para os agentes de baixo nível por meio do agente intermediário. Os agentes de baixo nível então atualizam suas visões. A comunicação entre agentes é sempre feita através do componente Controle de cada agente.

3.1.3.2 Conseqüências

O padrão de arquitetura *Presentation-Abstraction-Control* possui os seguintes benefícios:

- separação de funcionalidades promovida pelos agentes. Cada conceito semântico ou tarefa possui seus próprios dados (abstração) e interface (apresentação).
- Suporte à manutenção e extensão. Alterações na apresentação de um agente não afeta outros agentes e a inclusão de novas funcionalidades são adicionadas através da criação de novos agentes.
- Suporte à multi-tarefa. Agentes PAC podem facilmente ser distribuídos em diferentes *threads*, processos ou máquinas distintos.

Limitações do padrão:

- Aumento da complexidade do sistema. A divisão das funcionalidades do sistema em agentes PAC pode resultar em um sistema complexo. O projetista deve encontrar a granularidade correta das tarefas pois o número de tarefas indicará o número de agentes do sistema.
- O componente Controle é complexo, pois implementa a comunicação entre Abstração e Apresentação no agente e a comunicação com outros agentes do sistema.
- O *overhead* na comunicação entre agentes PAC pode ter impacto na eficiência do sistema, visto que se um agente de baixo nível recupera dados do agente de alto nível, todos os agentes intermediários existentes entre os dois são envolvidos na troca de dados.
- Aplicabilidade . Este padrão de arquitetura se aplica melhor a sistemas com conceitos semânticos atômicos maiores que necessitam de suas próprias interfaces particulares, como por exemplo: sistema de gerenciamento de tráfego em rede.

O projeto da arquitetura de sistemas utilizando PAC adapta-se bastante à abordagem de especificação de requisitos por meio de use-cases do método OOSE [Jac92]. Use-case provê a especificação do sistema por meio de casos de uso ou funcionalidades. [LM97] mostra um método para o desenvolvimento de interfaces que utiliza use-cases para a especificação do sistema e agentes PAC para a definição de sua arquitetura.

3.1.4 Comparação dos padrões de arquitetura

A tabela 3.1 mostra uma comparação entre os padrões de arquitetura descritos neste capítulo. A primeira linha da tabela compara a estrutura dos padrões. Os padrões MVC e MVC++ dividem uma aplicação interativa em 3 componentes únicos Modelo, Visão e Controlador. Já o PAC divide uma aplicação interativa em funcionalidades que serão representadas por agentes que possuem Apresentação, Abstração e Controle. A abordagem é bastante distinta. Quanto à independência de diálogo proporcionada às aplicações interativas, o MVC é o mais deficiente pois a comunicação entre Modelo e Visão é implementada diretamente nos componentes. O MVC++ possui um componente intermediário, o Controlador, entre Modelo e Visão. O PAC possui o Controle que separa Abstração e Apresentação. Ainda assim, não se pode dizer que o MVC++ e o PAC implementam de forma efetiva a independência de diálogo pois existem dependências entre a aplicação e sua interface, ainda que indireta. Quanto a complexidade, o MVC é considerado médio pois a separação dos componentes Visão e Controlador contribuem para torná-lo um pouco complexo, pois o comportamento dinâmico da interface está intimamente relacionado a ela. O PAC é considerado complexo caso a aplicação possua

pequenas e várias funções atômicas, neste caso existirão inúmeros agentes, cada um com sua própria interface. Quanto a eficiência, os padrões MVC e MVC++ são considerados razoáveis. O MVC perde em eficiência também com a separação entre Visão e Controlador. O MVC++ perde em eficiência ao implementar dependências entre Visão e Modelo indiretamente, por meio do componente Controlador. A eficiência do PAC é considerada baixa devido à complexidade do componente Controle, conforme visto na Seção 3.1.3.2. Finalmente, quanto a aplicabilidade, os padrões MVC e MVC++ são menos específicos, podendo ser utilizados em quaisquer sistemas interativos. Por outro lado, o padrão PAC se adapta melhor às aplicações que possuam conceitos semânticos atômicos maiores com interfaces particulares.

	MVC	MVC++	PAC
Componentes	**Model** – objetos da aplicação **View** – objetos visuais da interface gráfica. **Controller** – responsável pela interação com o usuário.	**Model** – objetos da aplicação **View** – objetos visuais da interface gráfica. **Controller** – conecta Modelo e Visão	**Presentation** – interface de um agente e interação com o usuário. **Abstraction** – objetos do modelo de um agente. **Control** – conecta os outros dois componentes e "conversa" com os outros agentes.
Abordagem	Por módulos: Modelo, Visão e Controlador	Por módulos: Modelo, Visão e Controlador	Funcional: procura-se identificar as tarefas atômicas do sistema.
Independência de Diálogo	Baixa	Média/Alta devido a existência do componente Controlador	Média/Alta devido a existência do componente Controle
Complexidade	Média – devido à separação de Controlador e Visão	Baixa	Alta – dependendo da característica do sistema.
Eficiência	Média – a separação dos componentes Controlador e Visão reduz um pouco a eficiência	Média - a existência do Controlador entre Modelo e Visão reduz a eficiência da comunicação entre interface e aplicação	Baixa – devido à complexidade do componente Controle, que, além de conectar Apresentação e Abstração em um agente, gerencia a comunicação entre agentes.
Aplicabilidade	Sistemas Interativos em geral	Sistemas Interativos em geral	Sistemas cujas tarefas atômicas não sejam pequenas e que possuam interfaces específicas.

Tabela 3.1: Comparação dos padrões de arquitetura MVC, MVC++ e PAC.

3.2 Frameworks Orientados a Objetos

A necessidade da reutilização de software é evidente. Os conceitos oferecidos pelas linguagens orientadas a objetos têm potencial para melhorar a reutilização de software de forma significativa se comparado à abordagem convencional. Existem dois tipos de reutilização de software [Lew95]:

1. Reutilização de componentes individuais
2. Reutilização de arquiteturas de software, isto é, vários componentes individuais organizados para trabalharem de forma colaborativa.

O caso 1 proporciona reutilização em fino grão. Os componentes individuais podem ser oferecidos em uma biblioteca de blocos de construção[7] . O caso 2 permite um nível bem maior de reutilização. Neste caso, reutiliza-se toda uma arquitetura algumas vezes pronta para o uso, outras vezes necessitando de extensões ou customizações para serem utilizadas. Neste trabalho, estamos interessados na reutilização de software em larga escala. Esta seção trata dos Frameworks orientados a objetos, uma forma de reutilizar software em larga escala no desenvolvimento de aplicações.

3.2.1 O que são Frameworks?

Frameworks, são coleções de classes colaborativas que capturam padrões e mecanismos que implementam os requisitos comuns de projeto em um domínio específico de aplicação [Fir93]. Os relacionamentos entre as classes provêem o nível certo de abstração para o consumidor do *framework*, o desenvolvedor de aplicação. A reutilização das interfaces internas do sistema e a forma como as funções são divididas entre os componentes do sistema são mais valiosas que a reutilização de código em si [JF88].

Uma classe abstrata é o projeto de um único objeto. Um *framework* é o projeto de um conjunto de objetos ou blocos de construção projetados para desempenharem de forma colaborativa um conjunto de responsabilidades. Diferentemente das classes abstratas isoladas, os *frameworks* permitem, além da reutilização de código, a reutilização de projeto, sendo considerados projetos de mais larga escala que classes abstratas [Tal94,JF88]. A contribuição do *framework* é a arquitetura que ele define para todas as aplicações que forem desenvolvidas a partir dele. *Framework* é uma idéia fortemente relacionada a arquitetura. Uma arquitetura é a forma como estão organizadas as partes de um sistema (subsistemas) e a forma como essas partes colaboram para formar o todo. Um *framework* é uma forma particular de representar arquiteturas, assim existem arquiteturas que não podem ser expressas como *frameworks* [KJ94].

---

[7] Do inglês *building blocks*

3.2.2 Frameworks x Bibliotecas de Classes

Frameworks, freqüentemente, são confundidos com bibliotecas de classe. Isso acontece porque algumas bibliotecas exibem comportamentos de *framework* e alguns *frameworks* podem ser utilizados como bibliotecas de classes. Entretanto, a diferença entre estes é clara. *Frameworks* podem ser considerados uma evolução das bibliotecas de classe em termos de reutilização. A Figura 3.9 mostra o uso de um *framework* em contraste com a Figura 3.10 que mostra o uso de bibliotecas de classes. Observe que a reutilização de projeto é a principal diferença entre os dois tipos de reutilização. A Tabela 3.2 resume as diferenças entre as duas abordagens.

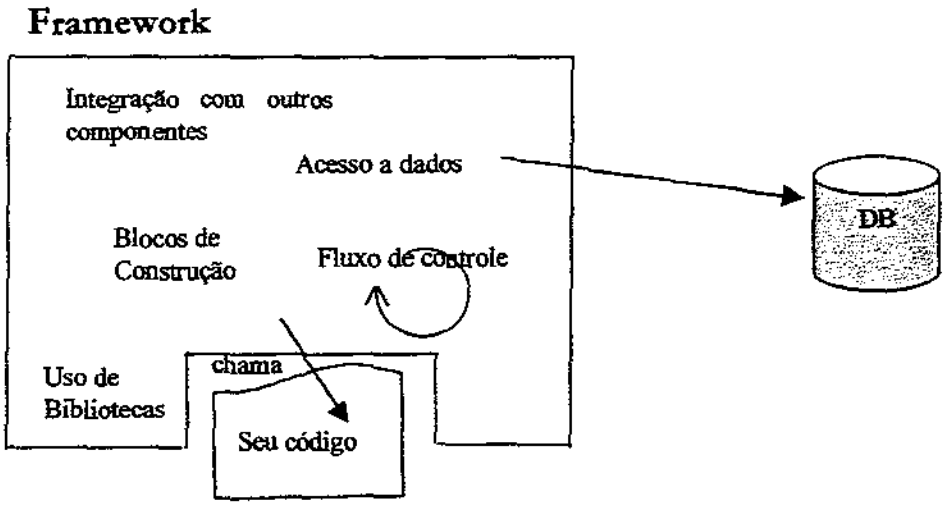


Figura 3.9 : Funcionamento de um framework.

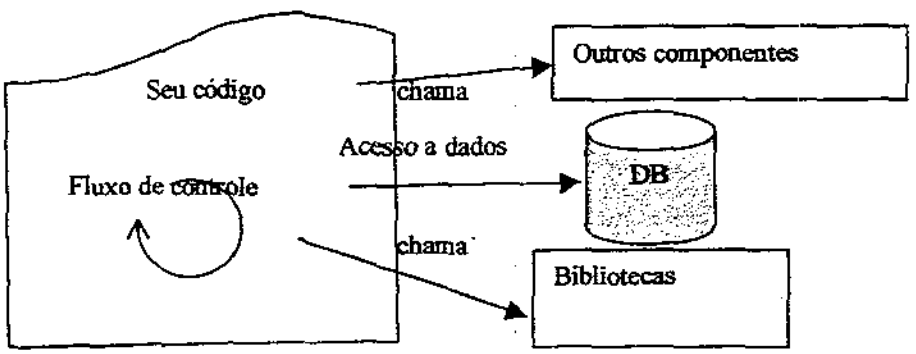


Figura 3.10: Uso de bibliotecas e componentes.

Bibliotecas de Classes	Frameworks
Provê reutilização através da instanciação de classes da biblioteca feitas pelo usuário	Provê reutilização através de subclasses criadas a partir de classes do *framework.*
Usuário chama as funções	Chama as funções do usuário
Não há fluxo de controle pré definido	Controla o fluxo de execução
Não há interação entre objetos pré definida	Define a interação entre os objetos
Não existe comportamento *default*	Provê comportamento *default*

Tabela 3.2: Bibliotecas de classes X *Frameworks*.

3.2.3 Classificação dos Frameworks

Os *frameworks* podem ser classificados segundo suas formas de construção e segundo o domínio do problema a que se referem. Os *frameworks* são divididos segundo suas formas de construção em *frameworks* caixa branca e *frameworks* caixa preta [Jfo88; Tal94]:

- **Framework caixa branca** ou dirigido a arquitetura[8] - para a utilização do *framework* o usuário precisa conhecer a forma como o mesmo trabalha internamente. Pois, para utilizá-lo, é necessário estender e modificar a funcionalidade, derivando novas classes e sobrecarregando algumas funções membro. A palavra-chave aqui é **herança**.
- **Framework caixa preta** ou dirigido a dados[9] - o desenvolvedor da aplicação precisa somente conhecer a interface externa dos componentes que fazem parte do framework e não a forma como elas colaboram entre si. Reutiliza-se o *framework*, instanciando classes e invocando suas funções membro. A palavra chave aqui é **composição**. Os *frameworks* caixa preta são semelhantes às bibliotecas de classes na forma de utilização, entretanto eles diferem das bibliotecas de classes no sentido de implementarem interação entre os objetos.

Frameworks caixa branca são mais flexíveis, portanto, atingem um número maior de situações onde podem ser reutilizados. No entanto, eles são mais difíceis de serem utilizados, porque é necessário saber como eles foram construídos. *Frameworks* caixa preta são menos flexíveis e mais fáceis de utilizar.

Quanto ao domínio do problema em que se aplica, Taligent classificou os *frameworks* em [Tal94]: *frameworks* de suporte, de aplicação e de domínio específico.

---

[8] Do inglês *architecture driven.*

[9] Do inglês *data driven.*

- **Frameworks de suporte** - provêem serviços de sistema, tais como acesso a arquivos, suporte a computação distribuída ou *device drivers*.
- **Frameworks de aplicação** - encapsulam o fluxo de controle da aplicação, como por exemplo os *frameworks* para a construção de aplicações com interfaces gráficas (*GUI frameworks - Graphical User Interface framework*).
- **Frameworks de domínio específico** - encapsulam o conhecimento de um problema particular, como por exemplo *framework* para multimídia.

Neste trabalho, tem-se interesse nos *frameworks* de aplicação caixa branca. Deste ponto em diante, o termo ***framework*** será utilizado para referenciar *frameworks* de aplicação caixa branca.

3.2.5 Por que utilizar Frameworks?

O uso de *frameworks* é incentivado pelo número de vantagens que ele proporciona ao desenvolvedor da aplicação, embora existam limitações que dificultam o uso efetivo dos mesmos. As subseções seguintes listam as vantagens e limitações no uso de *frameworks*.

3.2.5.1 Vantagens

Pode-se listar as seguintes vantagens no uso de *frameworks* [Orf96]:

- Estrutura pré-fabricada - *frameworks* reduzem codificação, depuração e teste, pois grande parte da estrutura da aplicação é oferecida pelo *framework*. Eles oferecem uma estrutura depurada, testada, pronta para o uso. Isso implica em um grande aumento da produtividade. Vale ressaltar que o código específico da aplicação e a colaboração entre o código específico e o *framework* devem ser depurados e testados.
- Guia arquitetural na construção de aplicação - na construção de uma aplicação não é necessário definir a arquitetura da aplicação.
- Redução de manutenção - quando várias aplicações são construídas a partir de um *framework*, tem-se uma redução dos custos de manutenção, pois se um erro é corrigido ou uma funcionalidade é adicionada ao *framework*, os benefícios ficam imediatamente disponíveis para todas as classes derivadas devido a herança, ou seja, os benefícios são disponibilizados para todas as aplicações construídas a partir do *framework*. Essa característica é muito importante, visto que os custos de manutenção de um software giram em torno de 60% a 85% do custo total de um software [Mey88].
- Padronização - um *framework* construído por meio de padrões de projeto e outros padrões organizacionais como estilo de codificação, por exemplo, permitem uma padronização das aplicações construídas a partir dele.
- Um *framework* oferece soluções de projetistas com experiência no domínio do problema a que se refere.

3.2.5.2 Limitações

Existem alguns fatores que limitam o uso de *frameworks* [Orf96]:

- Reutilização de software não acontece por acidente. Os projetistas do sistema precisam planejar a reutilização de seus sistemas, seja projetando software genérico para reutilização em diferentes situações específicas, seja generalizando um software específico para reutilização em outras situações. O tempo de desenvolvimento de um *framework* é bem maior que o tempo de desenvolvimento de uma aplicação específica. Este tempo extra deve ser visto como um investimento, cujo retorno é obtido a longo prazo, à medida que o *framework* for sendo reutilizado.
- A principal limitação no uso de *frameworks* é a curva de aprendizado. Quanto menos específico for o framework, maior deve ser o conhecimento sobre como as classes colaboram entre si [Joh92]. Dessa forma, ao se projetar um framework genérico, deve-se considerar a seguinte afirmação de Booch [Boo94]:

> "Um framework, por "mais elegante" que seja, nunca será utilizado, a menos que o custo do seu entendimento e o uso das suas abstrações sejam menores do que o custo do programador construir sua aplicação do início."

A Seção seguinte fala de técnicas utilizadas na construção e documentação de *frameworks*, as quais visam minimizar a dificuldade de entendimento do *framework*.

3.2.6 Construção e Documentação de Frameworks

Algumas técnicas são utilizadas na construção e documentação dos *frameworks*. Dentre elas, a mais utilizada é o uso de padrões de software, tanto na construção quanto na documentação do *framework*.

3.2.6.1 Construção de Frameworks

Padrões de software provêem uma abordagem promissora para o desenvolvimento de software com propriedades bem definidas [BMR96]. Eles são largamente utilizados na construção de *frameworks*. Padrões de arquitetura são utilizados na definição da arquitetura do *framework*, padrões de projeto são utilizados no projeto efetivo dos componentes do *framework* e da colaboração entre eles, e Idiomas são utilizados na implementação desses componentes. Dentre os padrões de software, padrões de projeto são os mais utilizados e mais importantes, pois a facilidade de entendimento do *framework* está diretamente relacionada à qualidade do projeto do *framework* em si. Uma vez definido o padrão de arquitetura, a parte mais trabalhosa na construção do *framework* é o projeto de sua arquitetura interna, ou seja, seus componentes e colaboração entre eles. Nesta fase, o uso de padrões de projeto facilita o trabalho e garante que as soluções utilizadas foram testadas e aprovadas por *experts* em projeto e desenvolvimento.

3.2.6.1.1 Padrões de Projeto e Frameworks

Freqüentemente, ocorre certa confusão nos conceitos de *frameworks* e padrões de projeto. Padrões de projeto são utilizados na construção e documentação de framework. Eles são elementos arquiteturais menores que *frameworks*, veja Figura 3.11, padrões são formados de classes e *frameworks* contém vários padrões. Padrões de projetos são mais abstratos que *frameworks*. O uso de um padrão de projeto em um *framework* é uma implementação específica do padrão [GHJ95].

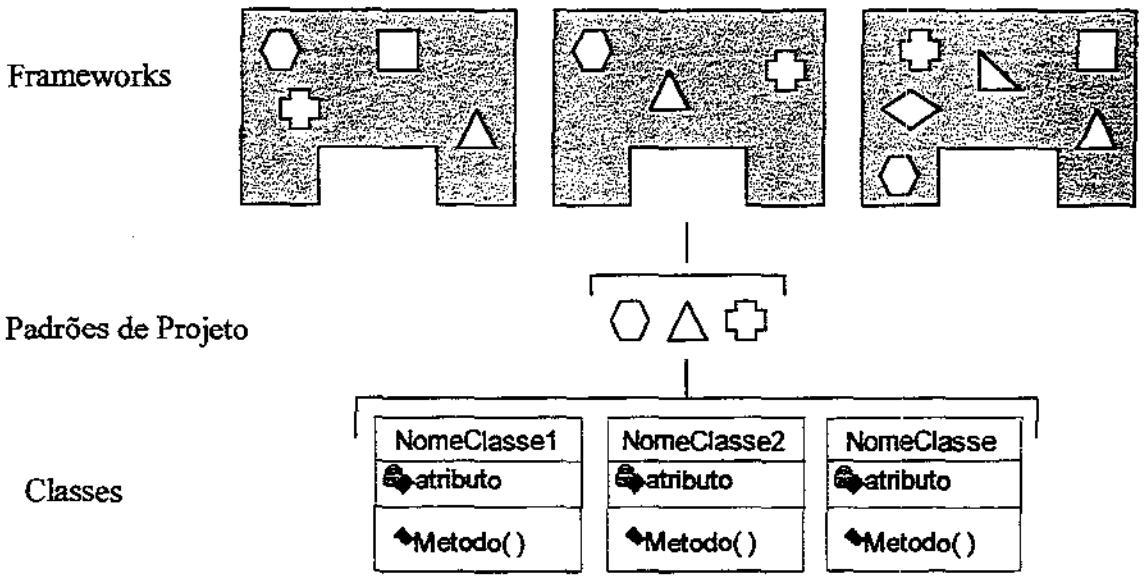


Figura 3.11: Os diferentes níveis de reutilização em um *framework*.

3.2.6.1.2 Metapadrões e Frameworks

Um *framework* fatora padrões recorrentes em algum domínio de aplicação. Existem padrões que não podem ser antecipados, por isso devem ser genéricos o bastante para serem facilmente adaptados às necessidades específicas de uma aplicação. Viu-se que metapadrões são padrões de projeto abstratos especialmente construídos para implementação de software genérico com partes flexíveis para serem adaptadas às características específicas das aplicações. Assim, metapadrões são úteis na implementação das partes flexíveis de um framework. Essas partes flexíveis do *framework* são chamadas **pontos adaptáveis** (*hot spots*), as quais devem ser implementadas por métodos acoplados dinamicamente. As partes não flexíveis são chamadas **pontos fixos** (*frozen spots*). Os métodos que implementam os pontos fixos são chamados *template* e os que implementam os pontos adaptáveis, métodos *hook*. O comportamento ou o inter-relacionamento entre os objetos são definidos pelos métodos *template* que invocam os métodos *hook* para adaptação do comportamento específico da aplicação [Pree94]. Observe que o padrão de interação básico do *framework* com o código da aplicação específico é definido pelos métodos *template*, e os pontos onde o usuário do framework deve inserir o código

específico da aplicação são os chamados pontos flexíveis implementados por métodos *hook*.

Na construção de um *framework* é mais fácil identificar pontos onde se encaixam metapadrões do que padrões de projeto, visto que os últimos são mais específicos. Metapadrões, ao contrário, são mais abstratos, adaptando-se mais facilmente às situações genéricas. No entanto, o uso correto de um padrão de projeto é mais fácil que o uso de um metapadrão. Para identificação da necessidade de um metapadrão e uso correto do mesmo é necessário uma certa experiência com os conceitos de orientação a objetos, pois é preciso identificar os métodos *template* e *hook* e implementá-los corretamente. Por outro lado, padrões de projeto, em geral, fornecem soluções mais terminadas cuja adaptação à uma situação específica é mais fácil, no entanto, menos freqüente de ocorrer por serem mais específicas.

3.2.6.2 Documentação de Frameworks

Visando a minimização da dificuldade de entendimento dos *frameworks*, alguns trabalhos têm sido desenvolvidos para a elaboração de métodos de documentação de *frameworks*. Johnson em [Joh92] propôs três tipos de documentação para *frameworks*: a que descreve o propósito do *framework*, o uso do *framework*, e o projeto do *framework*. Descrever o propósito do *framework* implica em descrever o domínio, os requisitos que ele satisfaz e qualquer limitação que ele possui A descrição do uso do *framework* aborda o *framework* do ponto de vista de um usuário, enquanto a descrição do projeto aborda a estrutura e comportamento do *framework* do ponto de vista do projetista do mesmo.

Dentre os trabalhos que enfocam a descrição do projeto e arquitetura, três abordagens são utilizadas. A primeira utiliza padrões de projeto - padrões de projeto são uma maneira natural de documentar um framework, visto que os mesmos são utilizados em sua construção. O uso de padrões de projeto conhecidos pode servir como um vocabulário comum entre o desenvolvedor e o usuário do framework [Joh92; KJ94]. A segunda fornece várias visões do projeto através de diagramas que descrevem os comportamentos estático e dinâmico de um sistema, bem como protocolos de interface e diagramas de configuração. A terceira utiliza exemplos - um exemplo consiste de uma implementação concreta de todas as classes abstratas e suas interações. Podem ser utilizadas ferramentas no rastreamento das interações e visualização de hierarquias de classes dos exemplos [GM95].

Os trabalhos que descrevem o uso do framework também utilizam três abordagens: Padrões de Projeto [Joh92] , Cookbooks [KP88] e Hooks [FHL97]. Padrões de projeto são usados para descrever os problemas com os quais o usuário irá deparar-se ao utilizar o *framework*. Um Cookbook foi primeiramente utilizado na descrição do propósito do *framework* Smalltalk MVC [KP88]. Ele mostra vários exemplos para ilustrar como os componentes devem ser utilizados. É similar a um livro de receitas ou a um tutorial. Hooks, assim como Cookbooks, provêem um guia para uso do *framework*. Ele é composto de hooks. Cada *hook* provê uma solução para um requisito específico na utilização do

framework. Em outras palavras, um *hook* ensina a implementar **pontos flexíveis** de um *framework*

3.2.6.3 Resumo

O programador da aplicação ao utilizar um *framework* deve conhecer sua interface e os mecanismos que ele implementa. Dessa forma, *frameworks* devem ser construídos de forma a facilitar o entendimento de seu projeto e a sua extensão de maneira que suas abstrações possam ser aplicadas a diferentes contextos. Por outro lado, sob o ponto de vista do projetista do *framework*, a construção e manutenção do próprio *framework* devem ser facilitadas. A separação das funcionalidades do *framework* em *subframeworks*, a implementação das dependências entre os *subframeworks* de forma transparente e, principalmente, a utilização de padrões de projeto na construção do *framework* são diretrizes para a construção de *frameworks*. Sob o ponto de vista do usuário, essas diretivas produzem *frameworks* mais fáceis de entender e utilizar, visto que os *frameworks* terão uma estrutura modular documentada com padrões de projeto conhecidos.

3.3 Exemplos de Frameworks para Interfaces Homem-Computador

Esta Seção descreve alguns dos *frameworks* mais conhecidos para desenvolvimento de aplicações interativas: o MacApp para desenvolvimento de aplicações Macintosh, o MFC para aplicações Windows e o ET++, para aplicações UNIX. Estes *framework* são baseados em uma variação do padrão de arquitetura MVC, o MV (Model-View). O MV é considerado a evolução do MVC, conforme visto na Seção 3.1.1.5.

3.3.1 MacApp

MacApp é um *framework* construído para desenvolvimento de aplicações Macintosh. Ele implementa a estrutura comum que todas as aplicações Macintosh compartilham, permitindo que o desenvolvedor da aplicação se concentre nas partes específicas da aplicação. A primeira versão do MacApp foi construída em 1987, ele foi o primeiro *framework* orientado a objetos disponível comercialmente. Existem no mercado, centenas de aplicações escritas utilizando o MacApp [Lew95].

A boa aceitação do MacApp se deve ao fato de ele simplificar extremamente o desenvolvimento de aplicações Macintosh. O Macintosh é conhecido pela consistência entre as interfaces de todas as suas aplicações. Porém a implementação de tais interfaces requer um árduo trabalho, pois o Macintosh possui centenas de funções úteis que caberia ao programador saber quais são apropriadas às suas necessidades. O MacApp simplifica este trabalho pois ele implementa o *loop* de eventos principal e distribui os eventos do usuário para os gerenciadores dos objetos da interface. *Clicks* de mouse na barra de menu são enviados como eventos para o Macintosh Menu Manager, por exemplo, cabendo ao programador escrever somente o código que processa o comando escolhido.

3.3.1.1 Estrutura

MacApp consiste de cerca de 180 classes. A Figura 3.12 mostra parte de sua hierarquia.
A hierarquia de classes possui uma única raiz, a classe *TObject*. *TEventHandler* representa um objeto que pode responder a vários eventos. Esses eventos podem ser oriundos do mouse, teclado ou qualquer evento definido pela arquitetura do MacApp. Um objeto *TCommand* define um comando iniciado pelo usuário através do menu por meio de teclado ou mouse. A classe *TCommand* encapsula os passos necessários à execução de um comando e armazena as informações suficientes sobre um comando para que ele possa ser desfeito através do comando *Undo*. Cada objeto comando tem um objeto *CommandHandler* associado, responsável por gerenciá-lo. A classe *Ttracker* , subclasse de *TCommand* implementa as ações do *mouse*. Isso implica que uma ação do mouse também pode ser desfeita por um comando *Undo*.

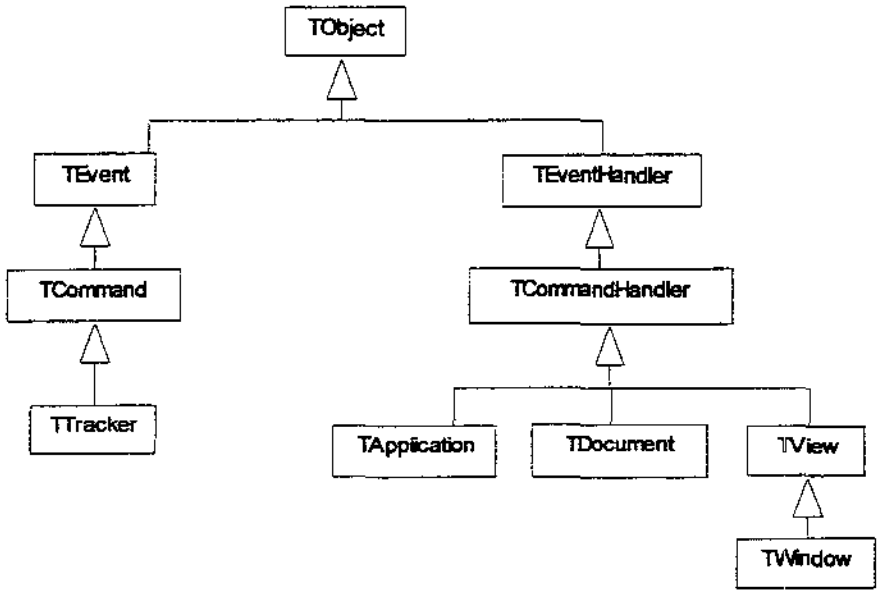


Figura 3.12: Hierarquia de classes parcial do MacApp.

A primeira classe que o usuário do MacApp deve conhecer é *TApplication*. Essa classe implementa o *loop* de eventos encontrado em todas as aplicações Macintosh, ela captura os eventos e os distribui para o apropriado tratador de eventos *(Application, Document, View, etc)*. Um objeto *TApplication*, por exemplo, trata eventos referentes a comandos de menu da aplicação tais como *Open*, *New* e *Quit*. *TDocument* define o comportamento geral dos documentos. Ela gerencia os dados do documento do usuário e processa comandos que lêem, escrevem e alteram os dados. *TView* define um sistema de coordenadas que limita a exibição dos desenhos e imagens para o usuário. Cada visão contém uma lista de subvisões. *TWindow* implementa o comportamento específico das janelas e é a interface entre a hierarquia de visões *MacApp* e o gerenciador de janelas *Macintosh*.

3.3.1.2 Interações

Para descrever como o *framework* trabalha internamente, Lewis at. all [Lew95] utilizaram uma aplicação simples desenvolvida no *framework*. A aplicação é um editor de ícone. A hierarquia de classes da aplicação é a seguinte:

TObject
  TEventHandler
    TCommandHandler
      TApplication
        TIconEditApplication
      TDocument
        TFileBaseDocument
          TIconEditDocument
      TView
        TIconEditView

As classes específicas da aplicação estão em negrito. A Figura 3.13 mostra as mensagens trocadas entre os objetos para criar um novo documento. *TApplication::Run* inicia o *loop* de eventos principal. O primeiro evento que a aplicação recebe é um evento do ambiente *Apple* que determina o que a aplicação deve fazer ao ser criada, como por exemplo criar um documento novo (*untitled1*) ou abrir um documento existente. *TAplication*, neste caso, cria um documento *TIconEditDocument* através do método *DoMakeDocument* que deve ser herdado e redefinido por *TIconEditApplication*.

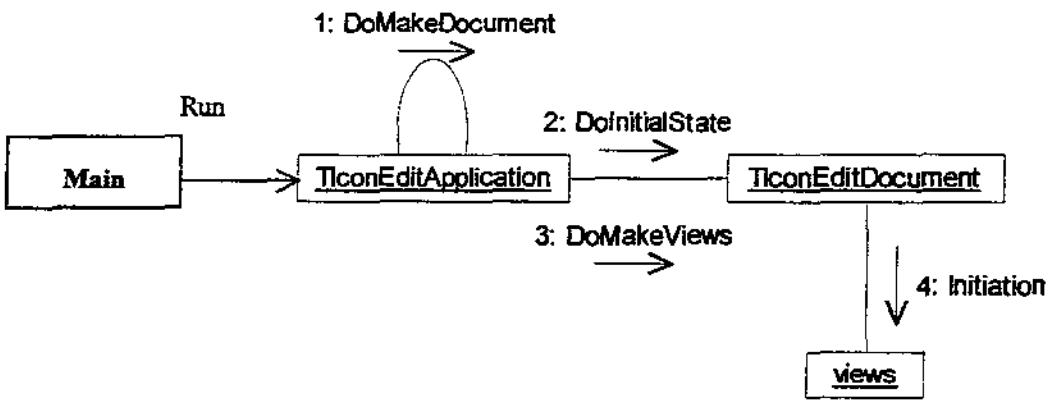


Figura 3.13: Iniciação de uma aplicação no MacApp.

DoInitialState é redefinido para a iniciação do *bitmap* com uma imagem *default*. Em seguida, *MacApp* chama *TDocument::DoMakeViews* e cria a janela com seus botões, barras de rolagem, e outros objetos visuais. *TIconEditDocument* herda o método *DoMakeViews* para criar uma janela que contém duas barras de rolagem e um *scroller*. O *scroller* contém uma instância de *TIconEditView*, que implementa somente o método *Draw* para copiar o *bitmap* armazenado em *TIconEditDocument* na tela (não mostrado na Figura). Para tanto, *Draw* chama *TIconBitmap::Draw* que por sua vez chama uma função da API QuickDraw implementada na *Toolbox Macintosh*, para efetivamente copiar o *bitmap*.

Uma das funcionalidades da aplicação é a inversão de cores do *bitmap*. Quando o usuário escolhe *Invert* na barra de menu, o objeto *TIconEditDocument* cria e inicia um objeto *TInvertCommand*, no método *DoMenuCommand*. *TIconEditDocument* chama *PostCommand* para a iniciação do processamento do comando. *MacApp* ao receber o objeto comando envia para ele a mensagem *DoIt* e altera o item de menu para *UndoInvert*. Se o usuário escolher *UndoInvert*, *MacApp* envia a mensagem *UndoIt* para o objeto comando e altera o item de menu para *RedoInvert*. A alteração da visão e a propagação da alteração para as outras visões são explicados na próxima Seção.

Observe, na Figura 3.12, que aplicação, documento e classes da visão são derivadas de *TCommandHandler*. Todas essas classes implementam o método *DoMenuCommand* para que os comandos referentes a essas classes sejam sejam tratados por elas especificamente. O *MacApp* usa um mecanismo de cadeia de prioridades para o processamento dos comandos do usuário. Se houver um documento aberto, a visão da janela ativa tem, primeiramente, a chance de processar o evento. Caso o evento não se aplique a ela, a mesma o repassa para o próximo tratador de comandos na cadeia. Esse processo se repete para cada tratador, até que se encontre o objeto responsável pelo tratamento do comando. O final da cadeia é o objeto *TIconEditApplication*. Se não houver documento aberto, a aplicação é a primeira tratadora da cadeia. Para o comando *InvertIcon*, *TIconEditView* tem a chance de primeiro tratar o comando, no entanto, ele o repassa para *TScroller*, através do método *DoMenuCommand*, que o repassa para *TWindow* que, por sua vez, repassa-o para *TEditDocument*. *TEditDocument::DoMenuCommand* é implementado para tratar o comando como dito acima.

3.3.2.3 Alteração da Visão e Propagação de Alteração

MacApp usa o padrão de arquitetura MVC na implementação da propagação de alteração das visões da aplicação (veja Figura 3.14). O método *InvertIcon* em *TIconEditDocument* causa uma alteração na respectiva visão *TIconEditView* (mensagem 3 da Figura). Então, *InvertIcon* chama o método *Changed*, definido no *MacApp*. *Changed* chama *DoUpdate* para todos os dependentes de *TIconEditDocument*. *DoUpdate* será então chamado para *TIconEditView* que chama *ForceRedraw* para fazer as alterações correspondentes. ForceRedraw chama *InvalidateRegion* de *Macintosh Toolbox* que envia a mensagem *Update* para a aplicação, se sua janela possui alguma área inválida (mensagens 4 e 5). *MacApp* processa a mensagem *Update* chamando *Update* de *Window* (mensagem 6). Isso faz com que seja redesenhada toda a hierarquia de visões através do método *Draw* de cada uma.

3.3.1.4 Resumo

O *MacApp* implementa o esqueleto básico de uma aplicação Macintosh de forma padronizada, ao mesmo tempo que fornece flexibilidade para que o usuário do *framework* adapte as características específicas de sua aplicação.

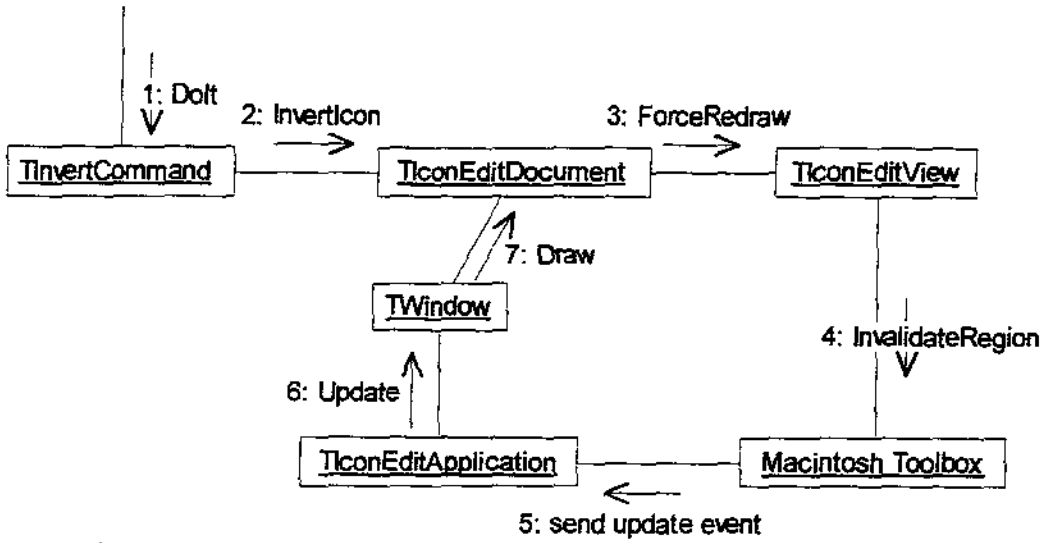


Figura 3.17: O funcionamento do mecanismo de propagação de alteração no MacApp.

3.3.2 Microsoft Foundation Class Library (MFC)

Microsoft Foundation Class Library (MFC) constitui um *framework* para interfaces gráficas implementado em C++ para o desenvolvimento de aplicações para ambiente Windows. Esta seção descreve o *framework* através de um exemplo, como na seção anterior [Lew95].

O MFC é utilizado juntamente com um ambiente de programação, o Visual C++ da Microsoft. O ambiente oferece ferramentas especialmente construídas para ajudar no uso do MFC: AppWizard e ClassWizard. AppWizard é usada apenas uma vez durante o desenvolvimento da aplicação, para gerar todas as classes de uma aplicação Windows *default* sem conteúdo, como menus, barras de rolagem, janelas, etc. ClassWizard facilita a tarefa de implementação dos tratamentos de eventos específicos. Ele também permite ao usuário a geração de novas classes e mapeamento das instâncias de variáveis para controlar itens de diálogos. ClassWizard é usado durante todo o desenvolvimento.

3.3.2.1 Estrutura

O MFC em sua hireraquia de classes provê:

- blocos básicos de construção (como as classes CObjList, CObjArray, CDate e CString);
- blocos gráficos de construção (como por exemplo *menus* e botões);
- componentes de alto nível da aplicação (como as classes CWinApp, CDocument, CView e CWnd).

A Figura 3.15 mostra a hierarquia de classes parcial do MFC. A hierarquia de classes possui uma única raiz, a classe CObject, de forma que todas as subclasses herdam seu comportamento. CObject implementa o armazenamento de meta-informação, para cobrir uma deficiência da linguagem C++ que não permite que o programador obtenha informações sobre os objetos. Assim, o MFC, como muitas bibliotecas implementadas em C++, implementam macros para a extração das informações necessárias das classes e objetos. MFC oferece acesso restrito à meta-informação através do método *IsKindOf(classname)*, que retorna verdadeiro se o objeto é uma instância de *classname* ou de uma de suas subclasses.

Cada aplicação rodando possui somente um objeto da classe *CWinApp*. A classe *CWinApp* obtém os eventos e os distribui para os outros componentes da aplicação. Uma outra funcionalidade é o gerenciamento (conceitual) dos documentos. Por exemplo, a aplicação é responsável por verificar se os documentos estão salvos, quando o usuário escolhe fechar a aplicação. As classes *CDocument* e *CView* correspondem aos componentes Modelo e Visão, respectivamente, do padrão de arquitetura MVC. Assim como no MVC, um documento pode ter várias visões. A Figura 3.16 mostra a estrutura de objetos de uma aplicação baseada no MFC.

```
CObject
    CCmdTarget
        CWinApp
        CDocument
        CWnd
            CFrameWnd
                CMDIFrameWnd
            CView
                CScrollView
            CControlBar
                CToolBar
                CStatusBar
            CDialog
            CButton
            CListBox
            CEdit
    CMenu
    CObjList
    CObjArray
```

Figura 3.18 : Hierarquia de classes parcial do MFC.

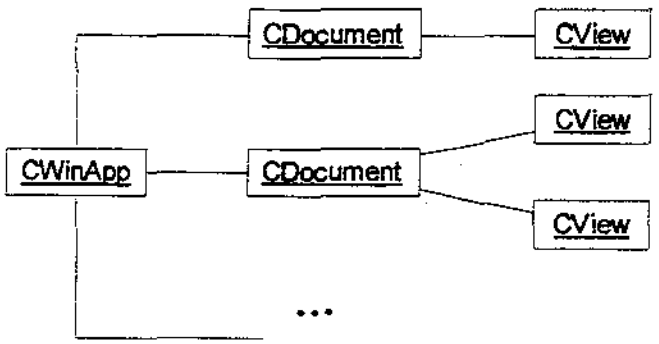


Figura 3.19: Estrutura dos principais objetos em uma aplicação baseada no MFC.

3.3.3.2 Tratamento de Eventos

O tratamento de eventos no MFC é semelhante ao implementado pela biblioteca convencional de funções estruturadas do Windows chamada Software Development Kit (SDK). Esta biblioteca implementa a leitura de eventos do sistema e chama funções do desenvolvedor da aplicação passando como parâmetro o evento lido para ser tratado. Essas funções devem implementar comandos *switch* para a identificação do evento. O MFC evoluiu a SDK encapsulando a identificação do evento, chamando métodos do programador da aplicação para o tratamento de eventos específicos. Se um objeto não trata determinado evento, o MFC o repassa para outros objetos, seguindo mecanismos baseados em estratégias específicas do Windows. Infelizmente, o MFC não utiliza característica de amarração dinâmica da linguagem C++, neste caso. Ele implementa uma amarração dinâmica através de comando de macros gerados pela ferramenta *ClassWizard*. Ganha-se em performance, talvez, mas a facilidade de leitura do código é altamente comprometida [Lewis95].

3.3.3.3 Características de Projeto

O *framework* MFC é ainda uma fina camada no topo da biblioteca SDK. Os usuários do MFC devem conhecer as mensagens do Windows para implementarem o tratamento de eventos da aplicação. Ou ainda, os programadores da aplicação lidam com o contexto de dispositivos de saída do Windows-SDK para implementarem as saídas gráficas. Em outras palavras, o usuário do MFC deve ser familiar à SDK também para desenvolver suas aplicações. Assim, o MFC é considerado um *GUI framework* em um estágio inicial pois ele facilita grandemente o desenvolvimento de aplicações Windows, possui algumas características dos melhores *GUI frameworks* ao mesmo tempo que faltam algumas características de abstrações mais avançadas tais como:

- MFC não implementa um pequeno *framework* para comandos serem desfeitos.

- O rolamento, divisão, ou aumento (*zoom*) do conteúdo de uma janela ainda requer grande esforço de implementação, se comparado a *frameworks* como o MacApp ou o ET++. Segundo Pree em [Lew95], a razão para a não implementação desta característica se deve ao fato de a abstração *CWnd* ser muito pesada para objetos visuais. Todas as instâncias de itens de controle herdam, por exemplo as características de *clipping* necessárias somente ao conteúdo da janela que é permitido rolar. O ET++, por exemplo, oferece uma solução mais leve e elegante para os objetos visuais, através da classe *VObject* e outros componentes como *Menu, Clipper, Scroller, Splitter e Zoomer*. Assim, não há uma classe única que implementa as várias características dos objetos visuais distintos.

Quanto a detalhes de implementação relacionados à linguagem, pode-se dizer que o MFC confunde o programador visto que o próprio *framework* implementa a amarração dinâmica referente a tratamento de eventos, através de macros. Desta forma, um objeto declarado estaticamente, por exemplo, pode ter métodos acoplados dinamicamente pelo MFC e métodos do C++, que são acopláveis dinamicamente, não são acionados devido a declaração estática do objeto.

3.3.2.4 Resumo

Apesar das deficiências do MFC, ele facilita o desenvolvimento de aplicações Windows se comparado ao uso da biblioteca de funções em C - SDK. Aplicações que possuem grande número de caixas de diálogo e acesso a base de dados são facilmente desenvolvidas devido ao auxílio proporcionado por ferramentas que acompanham o *framework*. No entanto, grande esforço de implementação é necessário se a aplicação possui recursos gráficos mais elaborados. Em suma, se for necessário entender os mecanismos de funcionamento do MFC, pode não ser tão simples utilizá-lo. Os iniciantes, em geral, confundem-se ao identificar o que é provido pelo *framework* e o que é provido pelo ambiente Windows.

3.3.3 ET++

O ET++ é um *framework* orientado a objeto para desenvolvimento de aplicações gráficas, implementado em C++ para o ambiente UNIX e sistemas de janelas como X11 e SUNWINDOW. Ele fornece "blocos de software" pré-fabricados para a construção da interface do usuário (e.g., janelas, barras de rolagem, menus, etc), estruturas de dados básicas, além das classes abstratas e concretas e relacionamentos necessários para a implementação de aplicações com interface gráfica em geral. Segundo Weinand et al. [WGM89], autores do *framework* , o uso do ET++ no desenvolvimento de uma aplicação pode reduzir em até 80% o tamanho código fonte, se comparado ao uso de outras *toolboxes* gráficas. Isto porque ele é um *framework* bastante completo que implementa grande parte do código comum às aplicações interativa em geral, como pode ser visto no capítulo 5.

3.4 Resumo

O uso de *frameworks* para interfaces homem-computador pode reduzir consideravelmente o tempo de desenvolvimento de uma aplicação interativa. As ferramentas de suporte ao desenvolvimento de interfaces, em geral, automatizam grande parte do desenvolvimento da interface gráfica, entretanto, a maioria dessas ferramentas não dá suporte à integração da interface com a aplicação. Ou melhor, não provê independência de diálogo nas aplicações interativas. Nos casos gerais são construídas aplicações fortemente acopladas à sua interface gráfica. Quando é usado um *framework* para interfaces homem-computador no desenvolvimento de uma aplicação interativa, o protocolo de comunicação entre interface e aplicação é herdado do *framework*. O grau de dependência entre os módulos é determinado pelo *framework*, mais precisamente pelo padrão de arquitetura no qual o *framework* foi baseado. Este capítulo descreveu os mais utilizados padrões de arquitetura para desenvolvimento de aplicações interativas e mostrou que nenhum deles proporciona uma independência de diálogo efetiva às aplicações interativas. Neste capítulo foram descritos ainda alguns dos mais conhecidos *frameworks* para o desenvolvimento de aplicações com interfaces gráficas. O próximo capítulo descreve o MVCR (Modelo-Visão-Controlador Reflexivo), uma combinação do padrão de arquitetura MVC com o padrão de arquitetura reflexão computacional, que visa proporcionar uma independência de diálogo efetiva para os *frameworks* para interfaces gráficas.

Capítulo 4

MVCR - Modelo Visão Controlador Reflexivo

O MVCR - Modelo Visão Controlador Reflexivo é uma combinação do padrão de arquitetura MVC, descrito na Seção 3.1.1, e o padrão de arquitetura Reflexão Computacional, descrito na Seção 2.5. Ele une a característica de modularização que MVC proporciona, dividindo uma aplicação interativa entre os módulos Modelo, Visão e Controlador, com a característica de separação dos aspectos de controle de comunicação entre interface e aplicação e o núcleo funcional da aplicação, proporcionada pela Reflexão Computacional. O resultado desta combinação é um padrão de arquitetura que possui módulos bem definidos completamente independentes, devido ao fato de o protocolo de comunicação entre os módulos ser implementado no meta-nível. As próximas Seções descrevem o padrão de arquitetura MVCR usando o formato de documentação de padrões: problema, solução e conseqüências, descrito na Seção 2.2.1.

4.1 Problema

O problema que o MVCR visa solucionar é o mesmo problema que motivou a criação do padrão arquitetural MVC: a grande dependência que existe entre núcleo da aplicação e interface gráfica da mesma. As seguintes forças continuam sendo necessárias serem asseguradas:

- A mesma informação é apresentada de diferentes formas. O sistema deve permitir que um dado da aplicação possua diferentes apresentações.
- Deve haver consistência entre os dados da aplicação e suas apresentações. A apresentação e comportamento do sistema devem refletir as manipulações de dados imediatamente. Existe uma conexão de causa entre a aplicação e sua apresentação (interface gráfica) que deve ser mantida em um sistema interativo.
- A interface deve dar suporte a diferentes *look and feel* padrões ou o porte da interface para uma outra plataforma não deve afetar a aplicação. Esse é um problema freqüente, pois as técnicas de interação com o usuário estão em constante evolução.
- A alteração da interface deve ser facilitada e possível até mesmo em tempo de execução.

O MVCR se propõe a resolver o problema abordando outros pontos que poderiam ser considerados também como forças.

- Os *Frameworks* para interfaces gráficas atuais são complexos de entender devido, em parte, ao acoplamento da interface e aplicação. Quanto maior a independência entre aplicação e interface gráfica, maior é a facilidade de compreensão do *framework* pois as partes podem ser analisadas separadamente.
- Adicionar uma interface gráfica a uma aplicação implementada não deve ser uma tarefa árdua. O padrão de arquitetura deve considerar a facilidade de adaptação de uma interface a uma aplicação já implementada. Isso deve acontecer independentemente da aplicação ser estruturada de acordo com o padrão arquitetural ou não.
- Componentes Modelo e Visão devem ser reutilizados independentemente sem esforço.

4.2 Solução

A facilidade de programar em um meta-nível, utilizando o padrão de arquitetura reflexão computacional, permitiu o surgimento de arquiteturas que visam proporcionar transparência para o programador de aplicações na resolução de problemas como tolerância a falhas [Lru96, LRB96], distribuição [CM93, Str92, TT89], concorrência [MWY91], persistência [SW88, Pae88], tipos de dados atômicos [SW95] e sonorização de aplicações [CPr96]. Esta prática reduz a complexidade da aplicação porque serviços administrativos são fornecidos de forma não intrusiva para a aplicação. Reflexão computacional provê uma separação mais clara entre as propriedades funcionais e não funcionais da aplicação [FNP95]. O *framework* para sonorização de aplicações [CPr96], por exemplo, permite que aplicações já existentes possam ser sonorizadas sem a necessidade de modificação dos seus códigos fontes. Todo o serviço de sonorização é fornecido de forma transparente para aplicação, não sendo necessário fazer qualquer alteração na aplicação para sonorizá-la ou retirar a sonorização da mesma. Analogamente, pode-se pensar em utilizar uma arquitetura reflexiva para adicionar uma interface a uma aplicação sem alterá-la. O fornecimento dos serviços de visualização da aplicação (interface gráfica) seria feito de forma transparente para a mesma.

A solução de projeto utilizando o padrão de arquitetura MVC é válida, pois é comprovada a qualidade da solução em termos de modularização de aplicações interativas. Considera-se um ponto a ser aprimorado, no entanto, a separação do protocolo de comunicação entre os componentes da funcionalidade do componente. Em outras palavras, o MVC não proporciona uma independência de diálogo efetiva e esta pode ser obtida se for feita um combinação dos padrões de arquitetura MVC e reflexão computacional. Essa variação do MVC denominou-se MVCR (Modelo-Visão-Controlador Reflexivo), veja Figura 4.1.

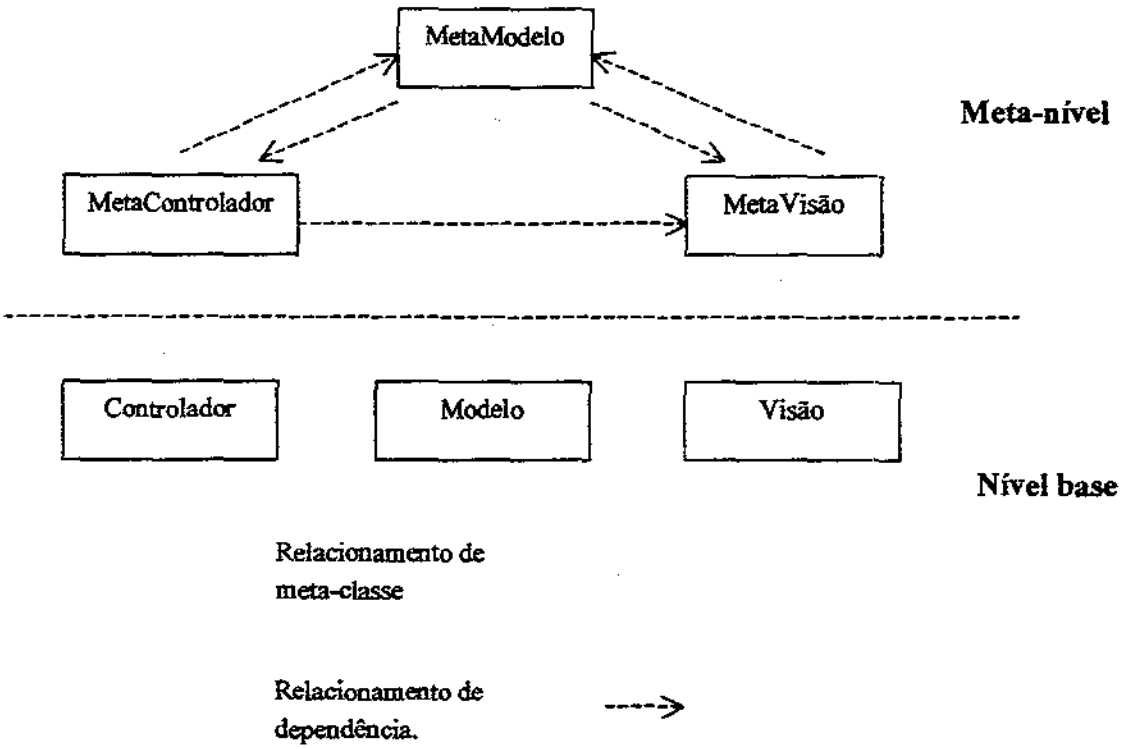


Figura 4.1: Arquitetura do modelo MVCR.

Os componentes do nível base implementam somente suas funcionalidades específicas. As dependências entre os componentes são levadas para o meta-nível. As próximas Subseções descrevem detalhadamente o padrão.

4.2.1 Estrutura Estática do Padrão

A Figura 4.2 mostra o padrão de arquitetura MVCR utilizando o padrão de projeto Observador para implementar o mecanismo de propagação de alteração entre os meta-objetos da aplicação. Observe, na Figura, que os aspectos não funcionais dos componentes são implementados no meta-nível. Tais aspectos não funcionais ou administrativos consistem do protocolo de comunicação entre os componentes, o qual possibilita a implementação da conexão de causa entre aplicação e sua interface gráfica. MetaModelo, MetaVisão e MetaControlador implementam as dependências entre os seus referentes Modelo, Visão e Controlador enquanto estes implementam a funcionalidade da aplicação, a interação com o usuário e a apresentação da aplicação, respectivamente. MetaVisão e MetaControlador observam o MetaModelo e MetaModelo está associado a zero ou mais observadores que são MetaVisões e MetaControladores.

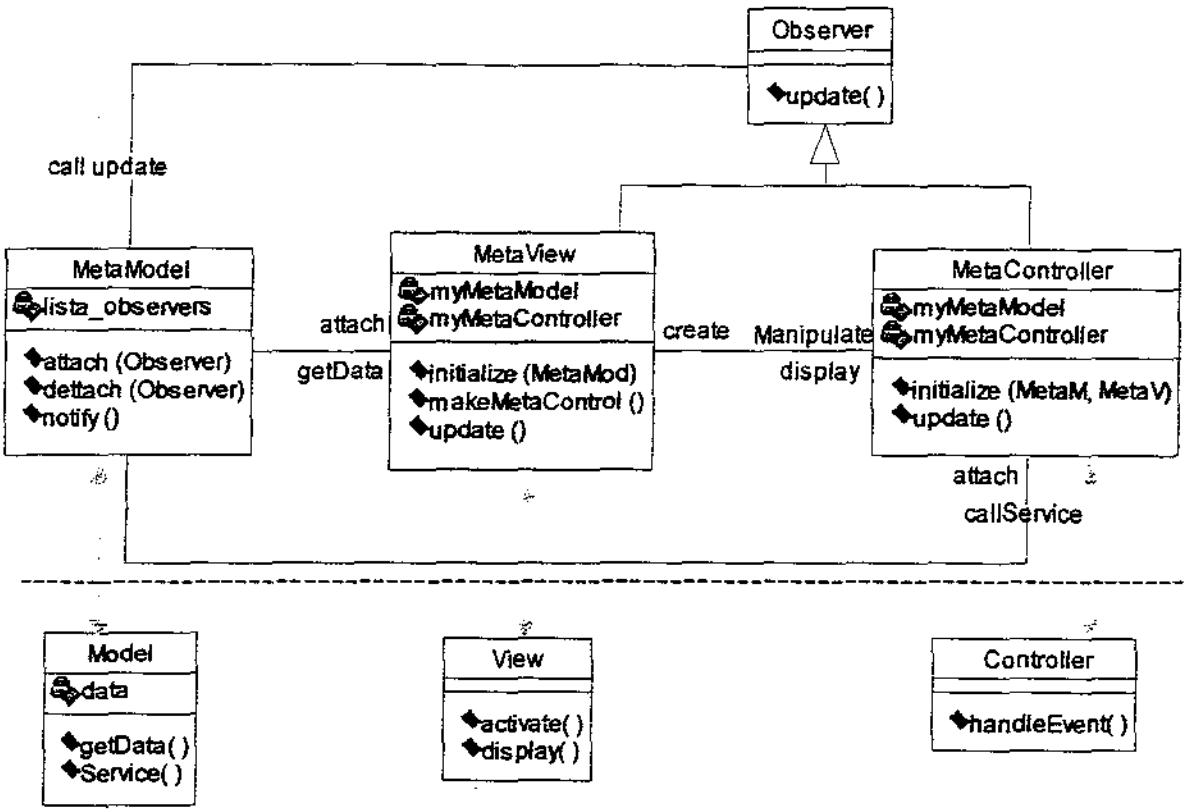


Figura 4.2: Implementação do MVCR utilizando o padrão de projeto Observador.

Classe Model :

A classe Modelo encapsula dados e exporta funções para acesso a esses dados e processamento dos mesmos. O usuário acessa às funções de processamento dos dados através do Controlador. Internamente à arquitetura o acesso às funções do Modelo acontecem via MetaModelo, para que o componente Modelo seja independente dos outros componentes. A Visão, por exemplo, utiliza as funções de acesso aos dados para manter atualizada a apresentação dos mesmos. Isso também acontece via MetaModelo, pelo mesmo motivo (veja Figura 4.2). A única responsabilidade do Modelo é:

- Implementar a funcionalidade da aplicação;

Classe MetaModel :

As responsabilidades da classe MetaModelo são:

- Registrar as MetaVisões e MetaControladores dependentes;
- Notificar os dependentes das alterações do Modelo.

O MetaModelo deve ser capaz de identificar uma alteração em seu referente (Modelo) e notificar seus observadores, a fim de manter a consistência do sistema. Isso é simples de implementar: basta que se identifique quais são os métodos que causam alterações no Modelo para que o meta-objeto controle tais métodos. Para usar o MVC, deve-se também

identificar esses métodos, a diferença, neste caso, é que quem se preocupa com as alterações e notificações é o meta-objeto e não o próprio objeto.

Classe View :
Diferentes Visões apresentam a informação ao usuário de diferentes formas. A responsabilidade do componente Visão é:

- Mostrar a informação da aplicação ao usuário;

As informação da Visão são obtidas através do componente MetaVisão.

Classe MetaView :
O mecanismo de propagação de alteração, disparado pelo MetaModelo, irá chamar o método *update* para as meta-visões dependentes. Cada MetaVisão implementará seu específico método *update*. Quando update é chamado, a MetaVisão deve recuperar os dados do Modelo via MetaModelo para atualizar seu referente: a Visão. O método *update* no meta-objeto implica que este deve alterar seu referente. Pode-se listar as seguintes responsabilidades para a MetaVisão:

- Criar e iniciar seus Controladores associados;
- Implementar o método *update;*
- Recuperar dados do Modelo.

Classe Controller:
O componente Controlador recebe as entradas do usuário como eventos, interpreta esses eventos, para transformá-los em requisições a serviços do Modelo e da Interface. Essas requisições são feitas através de seu meta-objeto, o MetaControlador. Pode-se citar como responsabilidades do Controlador:

- Reconhecer entradas do usuário como eventos;
- Traduzir eventos para requisição de serviços (*handleEvent*) – a serem efetuadas pelo MetaControlador.

Classe MetaController:
As responsabilidade da classe MetaController são:

- Implementar o procedimento *update*- muitas vezes o controlador é alterado quando determinado dado do Modelo é alterado, alguns serviços podem ser disponibilizados ou indisponibilizados de acordo com o estado do Modelo. Semelhantemente à MetaVisão, a implementação do método *update* faz alterações no referente, neste caso, no Controlador.

Observe que a estrutura mostrada na Figura 4.2 (padrão Observador) permite a implementação das responsabilidades listadas para cada componente. O padrão de projeto Observador foi descrito na Seção 2.3.

4.2.2 Comportamento Dinâmico

O comportamento dinâmico do padrão é mostrado através de diagramas de colaboração. O diagrama da Figura 4.3 mostra o processo de criação dos objetos e meta-objetos do sistema. Inicialmente, uma instância do Modelo e seu respectivo MetaModelo são criados (mensagem 1), em seguida, uma instância da Visão e sua respectiva MetaVisão (mensagem 2). Em seguida, a MetaVisão é iniciada com uma referência para o MetaModelo. A MetaVisão então cria o seu Controlador e MetaControlador e o inicia com as referências para o MetaModelo e para si própria (MetaVisão) (mensagem 3). Em seguida, o MetaControlador se inclui na lista de observadores do MetaModelo (mensagem 4). Finalmente, a MetaVisão também se inclui na lista de observadores do MetaModelo.

A Figura 4.4 mostra uma seqüência de execução que descreve uma alteração do Modelo disparando o mecanismo de propagação de alteração. Suponha que o acionamento de um botão do *mouse* tenha causado a execução do método *handleEvent* de Controller. O MetaController toma o controle de execução e requisita ao MetaModelo a execução dos serviços referentes ao acionamento do mouse. O MetaModelo, por sua vez, solicita ao Modelo os serviços. Suponha que a execução de tal serviço tenha causado alteração nos dados do Modelo. O MetaModelo toma o controle de execução, e executa o método *notify*. O método *notify*, chama *update* para seus dependentes: MetaVisão e MetaControlador. A Visão, na execução de *update*, recupera dados do Modelo, via MetaModelo, e solicita à Visão para mostrá-los ao usuário, semelhantemente, o MetaControlador também recupera dados do Modelo, via MetaModelo, para habilitar ou desabilitar alguma função de seu referente Controlador.

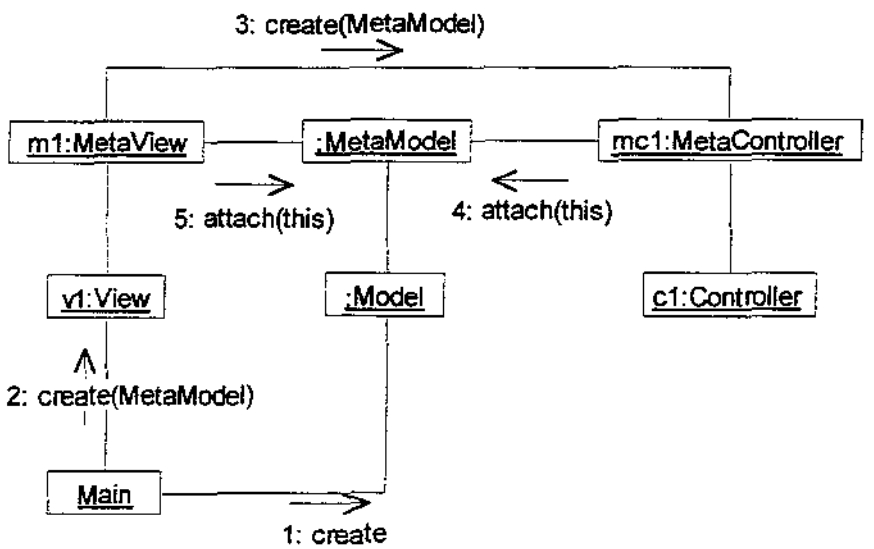


Figura 4.3 : Iniciação dos componentes do MVCR em uma aplicação.

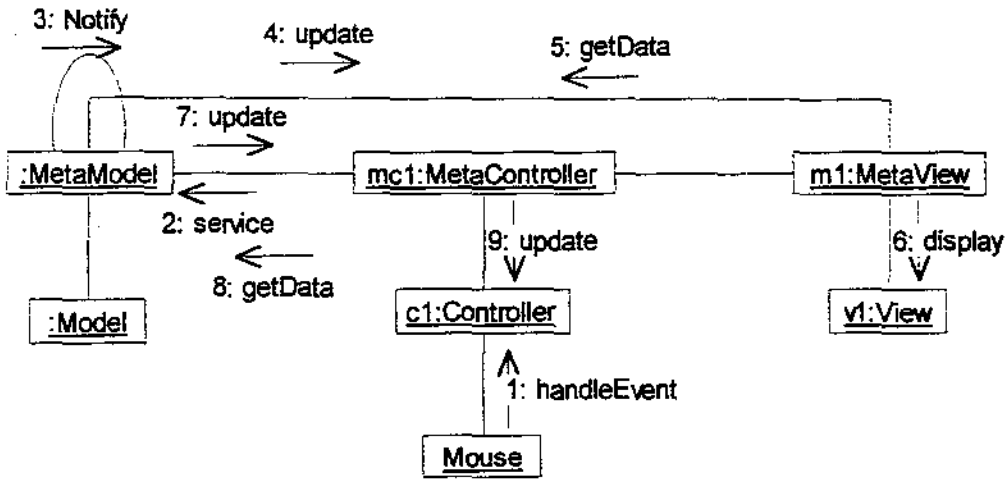


Figura 4.4: Funcionamento do mecanismo de propagação de alteração no MVCR.

4.3 Usos conhecidos

O MVCR ainda não foi utilizado na íntegra. Utilizamos uma simplificação do MVCR, o MVR (Modelo Visão Reflexivo), para reestruturar o ET++ resultando no ET++ Reflexivo. Como o ET++ é implementado utilizando o MV, fizemos uma reestruturação separando serviços funcionais e não funcionais seguindo a arquitetura do próprio *framework*. O capítulo 5 descreve o *framework* ET++ Reflexivo. A Seção 4.5 mostra a diferença básica entre o MVR e o MVCR.

4.4 Conseqüências

Esta Seção lista as vantagens e limitações do uso do MVCR.

4.4.1 Vantagens

Semelhantemente ao MVC, as vantagens da utilização do MVCR são:

- Múltiplas visões do mesmo modelo;
- Alteração de Visões e Controladores com pouco esforço.
- *Framework* potencial - é possível basear um *framework* de aplicação neste padrão.

Além das vantagens do MVC, têm-se outras, devido ao uso do meta-nível como outra dimensão de programação.

- A construção de aplicações complexas é facilitada, pois as tarefas de projeto e implementação podem ser divididas entre três grupos distintos de projetistas: os pro-

jetistas da interface gráfica, os projetistas da aplicação e os projetistas das dependências entre aplicação e interface gráfica. Os projetistas que implementam a interface gráfica (componente Visão) não precisam conhecer a interface da aplicação (componente Modelo) e vice-versa. Somente os projetistas que implementam as dependências entre os dois componentes (no meta-nível) precisam conhecer a interface de ambos.

- Um outro benefício é a possibilidade de reutilização dos componentes construídos a partir do *framework* independentemente. Uma aplicação já implementada poderia, sem grande esforço, ser acoplada a uma interface gráfica. Para tanto, alguns métodos da aplicação teriam de ser declarados como reflexivos e o MetaModelo implementado, além da interface (Visão e MetaVisão).
- entendimento do *framework* é facilitado quando o código de controle (dependência entre os componentes) são implementados no meta-nível e serviços da aplicação (os próprios componentes) no nível base.

4.4.2 Limitações

Semelhantemente ao MVC as desvantagens da utilização do MVCR são:

- Aumento da complexidade na separação de Visões e Controladores [WGM89].
- Ineficiência no acesso aos dados do Modelo na Visão.
- Dificuldade de utilizar o MVCR com modernas ferramentas de interface do usuário.

A seguinte desvantagens do MVC é atenuada pelo MVCR:

- O mecanismo de propagação de alteração causa uma alteração em todas as visões para uma simples ação do usuário. Muitas vezes, uma visão não necessita ser notificada de qualquer alteração, porque está minimizada, por exemplo – A existência de meta-objetos para controlar as alterações facilita a programação de mecanismos que controlem a propagação de alteração para a visão. As alterações do modelo que não afetarem a visão, por algum motivo, podem ser filtradas no meta-nível.

As seguintes limitações do MVC deixam de existir com o MVCR:

- Forte acoplamento entre visões e controladores que dificulta a reutilização de um ou outro separadamente.
- Forte acoplamento entre visões e controladores ao modelo – esse problema também não acontece com os componentes do nível base (Modelo, Visão e Controlador), somente com os do meta-nível, como pode-se observar pelo modelo estático da figura 2.

4.5 Uma abordagem reflexiva para o MV

A Seção 3.1.1.5 descreveu rapidamente o MV (*Model/View*) como uma evolução do MVC. Como dito na referida Seção, no MV o componente Controlador deixa de existir e suas atribuições passam a fazer parte do Componente Visão. Uma abordagem reflexiva pode também ser feita para promover uma maior independência de diálogo ao MV, assim como no MVC. O resultado seria um padrão mais simples que o MVCR devido a inexistência do componente Controlador. Do ponto de vista da comunicação entre os componentes Modelo e Visão, os padrões MVR e MVCR são equivalentes, desta forma a discussão das conseqüências de uso que é feita para o MVCR (Seção 4.5) é válida também para o MVR. A Figura 4.5 mostra uma interação com o usuário utilizando a abordagem reflexiva. Inicialmente, o usuário manipula a Visão, em seguida, a MetaVisão interrompe a execução das ações do usuário que afetam o Modelo. A MetaVisão solicita serviços ao MetaModelo. O MetaModelo solicita os serviços ao Modelo. A MetaVisão recebe os resultados e solicita serviços de apresentação dos resultados à Visão.

A vantagem da utilização de reflexão, neste caso, é também a independência dos componentes obtida na implementação. Isso acontece porque a comunicação entre os componentes do nível base não envolve associações ou composições, esta é feita por meio de interrupções efetuadas pelo meta-nível.

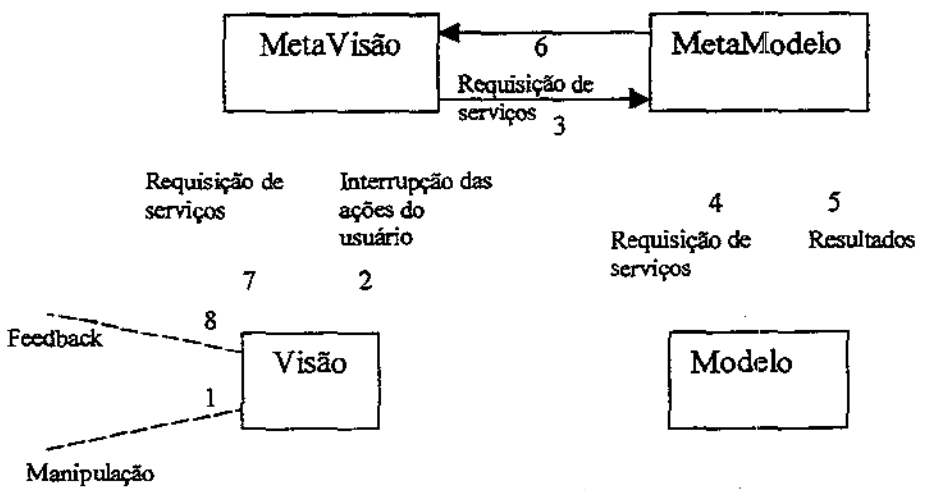


Figura 4.5 : O padrão MVR como uma simplificação do MVCR.

4.6 Uma abordagem reflexiva para o MVC++

O MVC++, como visto na Seção 3.1.2, procura melhorar a independência de diálogo no MVC através da existência do componente Controlador que separa os componentes Modelo e Visão e não mais implementa a interação com o usuário. Uma abordagem reflexiva para o MVC++ induz a um padrão idêntico ao MVR (Figura 4.5). Como no MVC++, o componente Controlador implementa apenas a integração dos componentes Modelo e Visão, uma reestruturação deste padrão utilizando reflexão computacional levaria todo o componente controlador para o meta-nível, como mostra a Figura 4.5. Ou seja, o componente Controlador seria representado pelos componentes MetaModelo e MetaVisão descritos neste capítulo. A descrição de uma interação com o usuário nesta abordagem reflexiva para o MVC++ seria idêntica à interação descrita para o padrão MVR na Figura 4.5.

4.7 Resumo

Este capítulo descreveu o MVCR (Modelo Visão Controlador Reflexivo) como uma proposta de um padrão que proporciona maior independência de diálogo aos sistemas interativos. Foram apresentadas também abordagens reflexivas para o MV e para o MVC++ com o objetivo de mostrar a possibilidade do uso de reflexão computacional na construção de sistemas interativos combinado com outros padrões de arquitetura. Coincidentemente a abordagem reflexiva para o MV é similar à abordagem reflexiva para o MVC++, isto acontece porque o MVC++ é o padrão MV com um novo componente Controlador que visa obter independência de diálogo entre Modelo e Visão e na abordagem reflexiva do MVC++, esse componente Controlador é modelado no meta-nível; semelhantemente ao MVR que leva as dependências entre Modelo e Visão para o meta-nível (obtendo-se um componente Controlador que conecta Modelo e Visão no meta-nível). O capítulo seguinte mostra um experimento prático utilizando o *framework* ET++ com o objetivo de comparar o MVC e o MVCR.

Capítulo 5

Estudo de Caso: MVC versus MVCR

Este capítulo mostra um estudo comparativo entre a abordagem convencional do MVC e a abordagem reflexiva do MVCR. Utilizamos a implementação de uma mesma aplicação nas duas abordagens para avaliar as vantagens e desvantagens da abordagem reflexiva. A aplicação é desenvolvida primeiramente a partir do *framework* ET++, que é baseado em uma simplificação do padrão de arquitetura MVC – o MV (Seção 5.1). Posteriormente, a mesma aplicação é desenvolvida a partir do ET++ Reflexivo, uma reestruturação do ET++ utilizando uma simplificação do padrão de arquitetura MVCR - o MVR (Seção 5.2). Como visto anteriormente, o MV (Modelo-Visão) é considerado por muitos uma evolução do MVC. O MVR (Modelo Visão Reflexivo) é descrito como a combinação do padrão de arquitetura MV com o padrão de arquitetura Reflexão Computacional (Seção 4.5). As características básicas do MVCR são preservadas, a diferença básica é a seguinte: como o MV possui somente Modelo e Visão, o MVCR possui somente MetaModelo e MetaVisão. A Seção 5.3 faz uma comparação entre os estudos de caso das Seções 5.1 e 5.2. As conclusões do estudo de caso podem ser tidas como conclusões para uma comparação entre o MVC e o MVCR pelo fato de que a interação entre os componentes Modelo Visão e Controlador através de componentes no meta-nível no MVCR é semelhante à interação entre os componentes Modelo e Visão também através de componentes no meta-nível no MVR. A solução do MVR é uma simplificação do MVCR que reduz a complexidade de implementação da mesma forma que o MV reduz a complexidade do MVC. No aspecto independência de diálogo (problema que motivou o desenvolvimento deste trabalho), que se refere à comunicação entre Modelo e Visão, os padrões MVR e MVCR são equivalentes, portanto a utilização de um *framework* baseado no MV e não no MVC não traz nenhum prejuízo ao experimento.

5.1 ET++

Como dito anteriormente, o ET++ é um *framework* orientado a objetos para desenvolvimento de aplicações gráficas, implementado em C++ para o ambiente UNIX. Como dito anteriormente, a arquitetura do ET++ foi bastante influenciada pelo Smalltalk-80 [Gold83], um *framework* para o desenvolvimento de aplicações, utilizado na construção do ambiente de programação de Smalltalk. A hierarquia de classes do ET++ possui uma única raiz, a classe *Object*, assim como em Smalltalk-80. Essa abordagem resulta em um

sistema mais homogêneo e permite que todas as classes herdem funcionalidades básicas implementadas nesta classe. Uma outra influência do Smalltalk-80 é o uso de uma variação do padrão de arquitetura MVC (Model/View/Controller), descrito na Seção 3.1. Como dito anteriormente, o ET++ implementa o modelo MV, uma simplificação do MVC, que separa Modelo e Visão e divide o Controlador entre esses dois componentes.

5.1.1 Estrutura

O ET++ fornece "blocos de software" pré-fabricados para a construção da interface do usuário (e.g., janelas, barras de rolagem, menus, etc), estruturas de dados básicas, além das classes abstratas e concretas e relacionamentos necessários à implementação de aplicações com interface gráfica em geral. A Figura 5.1 mostra a estrutura interna do ET++; seus componentes são descritos nos itens a seguir.

5.1.1.1 Blocos de software básicos

Os blocos de software básicos definem estruturas de dados básicas de uso geral como conjuntos, listas, arranjos, etc. Essas estruturas manipulam quaisquer objetos derivados da classe *Object*. O ET++ fornece também mecanismos para manipular e inspecionar os elementos das estruturas um por um, são os chamados Iteradores (*Iterators*) robustos.

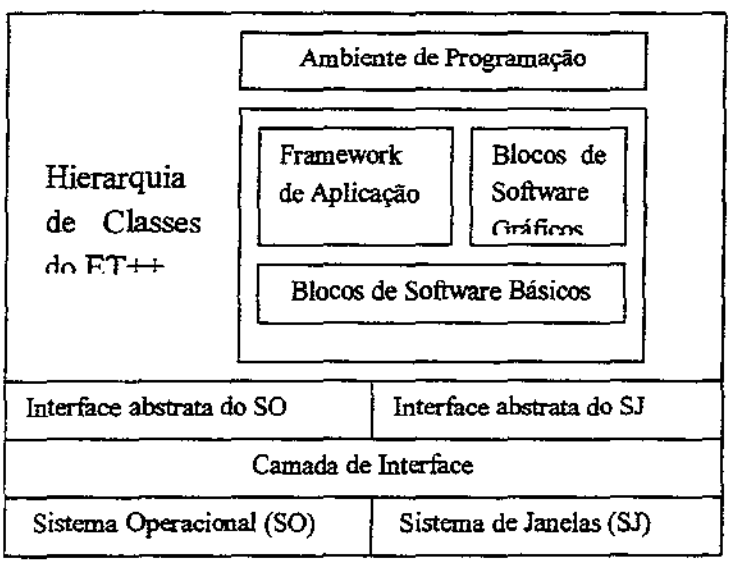


Figura 5.1: Arquitetura do ET++.

5.1.1.2 Blocos de Software Gráficos

Os blocos gráficos de construção definem os componentes gráficos e interativos, tais como janelas, menus, caixas de diálogo, barras de rolagem, etc. Esses blocos gráficos estão disponíveis em quase todas as ferramentas para construção de interface interativa. O ET++, diferentemente da maioria dessas ferramentas, fornece um *framework* que facilita a

modificação e construção de novos blocos a partir dos existentes. Os objetos gráficos no ET++ são todos derivados da classe *VObject*, veja a hierarquia de classes na Figura 5.2. Esta classe define um protocolo abstrato para gerenciar o tamanho da figura, a imagem dos objetos na tela e o tratamento de eventos. Para criar uma subclasse, é necessário redefinir um pequeno número de métodos. O restante da Figura 5.2 é explicado na Seção 5.1.4.

Um mecanismo interessante implementado no ET++ é a facilidade de combinação de vários objetos visuais em um único, através da classe *CompositeVObject*. Dessa forma, objetos visuais compostos são tratados como um único objeto. A classe *CompositeVObject* aplica as suas operações a todos os objetos que compõem o objeto composto e também repassa para eles os eventos recebidos.

5.1.1.3 Camada de Interface do Sistema

Essa camada proporciona independência das classes do ET++ em relação ao sistema operacional e ao sistema de janelas, garantindo assim a portabilidade do *framework*. Todas as dependências do sistema foram encapsuladas por um conjunto de classes abstratas que fornecem um conjunto mínimo de funcionalidades de baixo nível, necessárias para implementar o ET++ e consequentemente as aplicações desenvolvidas a partir dele. Dentre essas funcionalidades podem-se citar:

- serviços do sistema operacional;
- fonte, cursor, gerenciamento de imagens;
- funções gráficas, gerenciamento de janelas e manipulação de entrada de dados[10].

As interfaces abstratas do Sistema Operacional e Sistema de Janelas mostradas na Figura 5.1 são implementadas pela classes abstratas *System* e *WindowSystem*, as quais definem o ponto de entrada do ET++ para as classes de interface do sistema. Essas classes possuem subclasses concretas que provêem implementações específicas para os sistemas operacionais e sistemas de janelas. A instanciação do sistema operacional e de janelas corretos é feita em tempo de execução da aplicação.

5.1.1.4 Classes do *Framework* de Aplicação

O ET++ fornece um projeto abstrato para um tipo particular de aplicação através de uma coleção de classes abstratas. Dentre as classes podemos citar: *Application*, *Document*, *Manager* e *EvtHandler*. Essas classes implementam o fluxo de controle de uma aplicação genérica com interface gráfica. Elas implementam o protocolo de comunicação da aplicação com as classes mais genéricas da interface gráfica: *VObject* e *View*. A hierarquia de classes do *framework* de aplicação pode ser visualizada na Figura 5.2. As principais decisões de projeto do ET++ são descritas nesta Seção por meio de metapadrões e padrões de projeto.

---

[10] Do inglês *input handling*

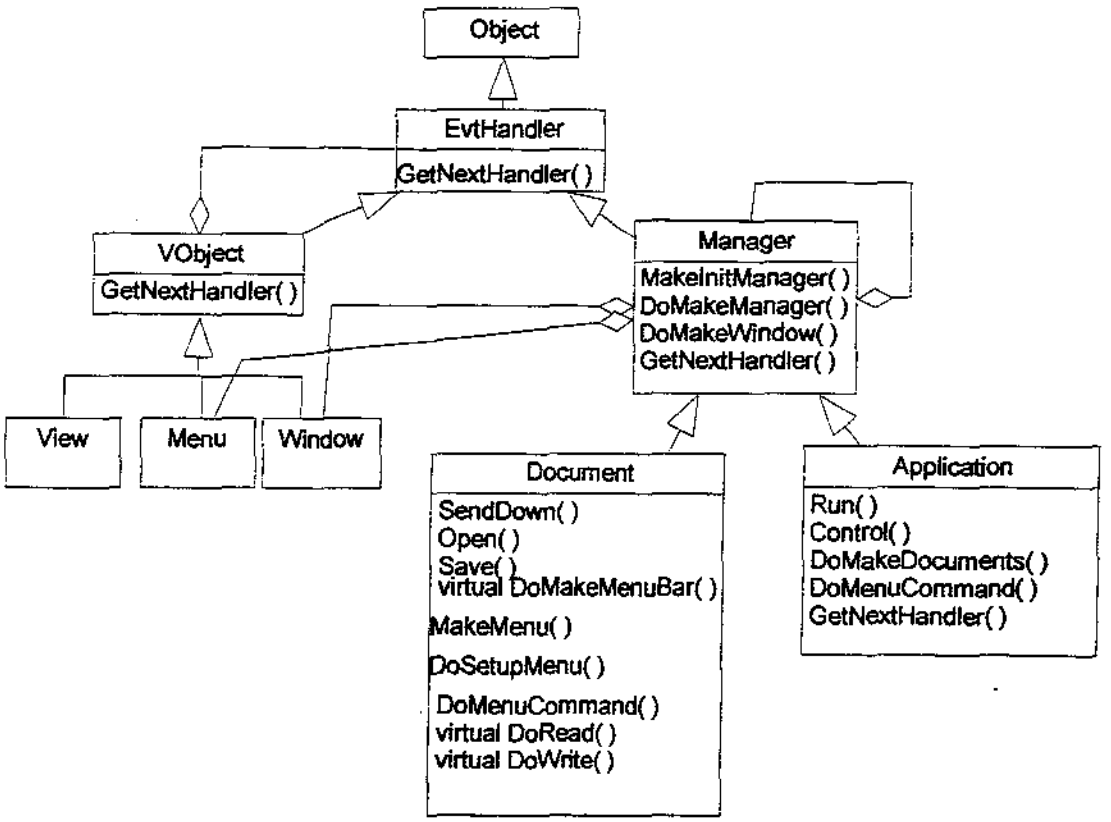


Figura 5.2: Hierarquia de classes do *framework* de aplicação.

Classe Object

A classe *Object* é raiz de quase todas as classes do ET++, somente as classes usadas para construção de outros tipos como *Rectangle* e *String* não descendem de *Object*. Dentre as funcionalidades implementadas em *Object* pode-se ressaltar: comparação entre objetos, propagação de alterações nos objetos para os objetos dependentes (notificação de objetos) e suporte para a transferência (entrada e saída) de estruturas de dados complexas através da implementação dos métodos *ReadFrom* e *PrintOn*.

A propagação de alteração é implementada através de uma variação de implementação do padrão de projeto Observador, discutido no capítulo 2. A implementação do Observador mostrada na Seção 2.4 é equivalente à implementação do metapadrão 1:N Conexão mostrado na Seção 2.5. No ET++, a classe *Object* faz ambos os papéis de Observador e Observado, nesta implementação o padrão Observador é equivalente ao metapadrão 1:N Unificação Recursiva, veja Figura 5.3. Isso implica que uma hierarquia de objetos dependentes pode ser definida, ou seja, objetos que observam um objeto em particular podem, por sua vez, serem observados por outros objetos. Neste caso, deve-se tomar cuidado com dependências circulares, pois elas causam um laço infinito. O método *template* é o método *Send* e o *hook* é o método *DoObserve*. A fim de notificar uma alteração em um objeto, o método *Send* é invocado, *Send* invoca *DoObserve* para todos

os seus observadores. *DoObserve* deve ser redefinido por cada observador, para efetuar as alterações correspondentes à alteração no objeto observado.

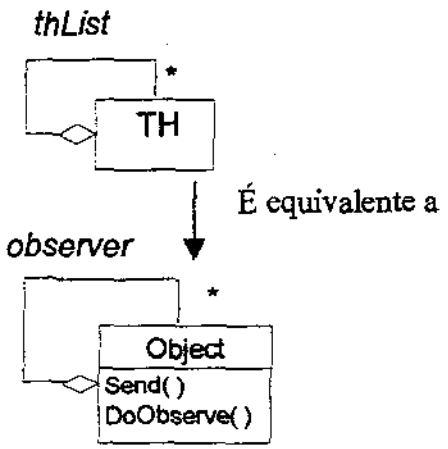


Figura 5.3 : Propagação de alteração baseada no metapadrão 1:N Unificação Recursiva.

Classe Manager

Esta classe implementa o gerenciamento das aplicações em geral. Esse fluxo de controle são *frozen spots* (pontos fixos). Como vimos anteriormente, os pontos fixos são implementados por métodos *template* que invocam métodos *hook* para adaptar o comportamento específico das aplicações. Neste caso, os métodos *hook* são responsáveis pela instanciação da aplicação específica, dos documentos específicos e dos objetos que representarão a aplicação graficamente. Cada aplicação ET++ possui apenas um objeto *Application* que gerencia qualquer número de documentos. O projeto da classe *Manager* é representado pelo metapadrão **1:N Unificação Recursiva**, como mostra a Figura 5.4. O conceito de *Manager* é aplicado recursivamente: um objeto *Manager* gerencia zero ou mais objetos *Manager*. Considerando que as classes Application e *Document* são subclasses de *Manager*, a situação em que um objeto *Application* gerencia um número arbitrário de documentos é um caso especial deste projeto. Neste caso, uma aplicação é o *Manager* da instância de *Document* criada a partir dela e *Document*, por sua vez, é o *Manager* dos objetos visuais que ele possui.

Um menu e uma janela, criados pelo respectivo *Manager* de cada um, estão associados a cada documento

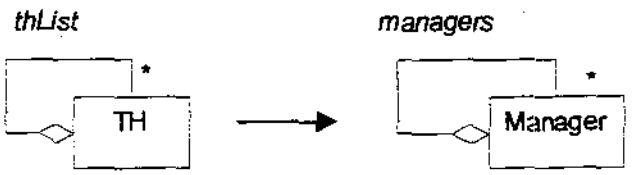


Figura 5.4: Projeto da classe *Manager* representado pelo metapadrão 1:N Unificação Recursiva

Classe EvtHandler

A classe *EvtHandler* é responsável pela distribuição e tratamento de eventos no ET++. Os eventos emitidos pelo usuário da aplicação, captados pela interface gráfica, são passados para a aplicação e para o documento (componente modelo) através da classe *EvtHandler*. Ambos, modelo e visão são derivados de *EvtHandler*, pois ambos manipulam e reagem aos eventos em geral. Observe na Figura 5.5 que um objeto *EvtHandler* possui uma referência para o próximo objeto *EvtHandler*, formando uma cadeia em um grafo direcionado, no momento da execução. Essa cadeia é alterada e muda dinamicamente de acordo com a execução da aplicação. Um objeto *Application* recebe todos os eventos de entrada e os envia para a cadeia apropriada de tratadores de eventos. Através do método *SetFirstHandler*, o programador da aplicação define qual objeto em uma cadeia particular deve receber o evento primeiro. Caso o *FirstHandler* não saiba tratar o evento, ele o repassa para o próximo *EvtHandler* na cadeia. Por exemplo, suponha a cadeia de tratadores de eventos de uma execução de uma aplicação que possui documentos, mostrada na Figura 5.6. No exemplo, existem dois *Documentos* abertos. A classe *MyApplication* é subclasse da classe *Application* e a classe *MyDocument* é subclasse de *Document*. Um objeto *MyDocument* usa um objeto *VObjectTextView* para mostrar e editar seus dados. Para reagir aos eventos, as classes devem herdar os métodos correspondentes aos eventos. Para reagir a uma seleção de um item do menu, por exemplo, o método *DoMenuCommand* deve ser redefinido. Existe uma definição padrão para este método na classe *Document*, qualquer comportamento particular deve ser implementado na subclasse *MyDocument* ou até mesmo na classe *VObjectTextView*.

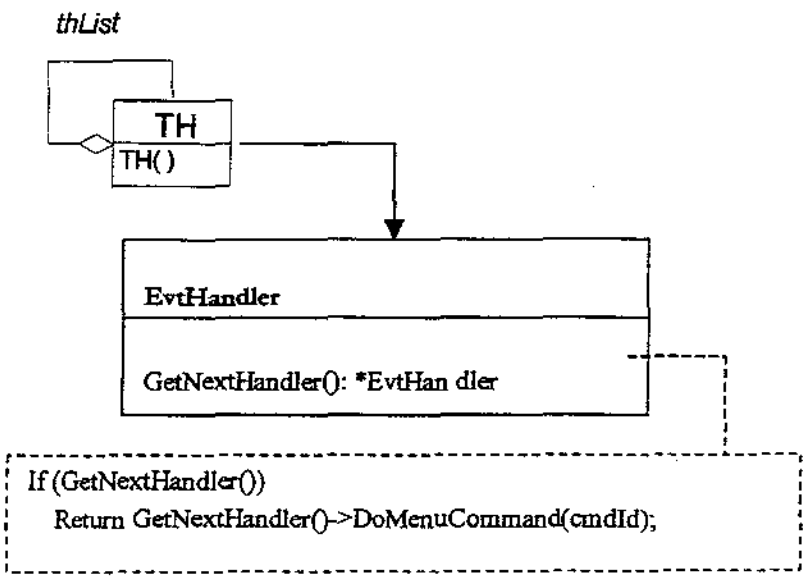


Figura 5.5: Classe *EvtHandler* baseada no metapadrão 1:1 Conexão Recursiva.

Classe Application

Uma aplicação genérica do ET++ consiste de uma instância da classe *Application*. A classe *Application* implementa as operações básicas que são comuns às operações em geral como: *Run* e *Quit*. As operações comuns aos objetos documentos como fechar e minimizar são implementadas na classe *Manager* com fins de reutilização.

A classe *Application* é também responsável pela instanciação dos documentos através do seu método *hook DoMakeDocuments()*, invocado pelo método *template DoMakeManager()* da classe *Manager*.

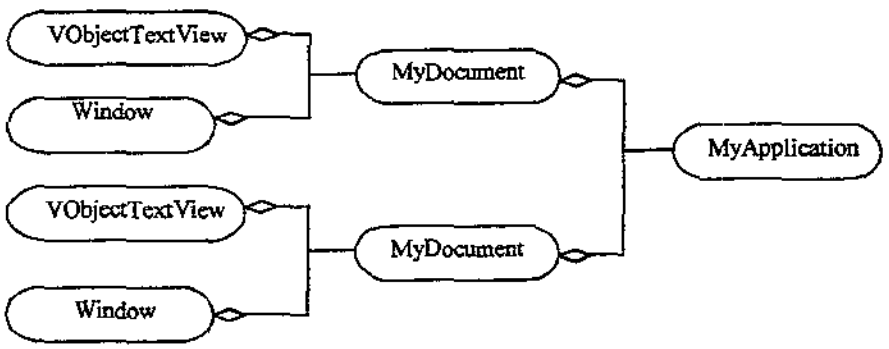


Figura 5.6: Exemplo de uma cadeia de tratadores de eventos em uma aplicação.

Classe Document

Document é a classe do componente modelo que representa documentos em geral. As operações básicas, efetuadas com documentos são implementadas genericamente por meio de métodos *hook* que devem ser redefinidos nas subclasses concretas. Operações como abrir, fechar e salvar são implementadas por métodos template que invocam os métodos *hook DoRead* e *DoWrite*, os quais devem ser herdados e definidos na subclasse de *Document*. Esses métodos são dependentes da aplicação pois dependem do tipo do dado o qual o Documento irá manipular.

Esta classe implementa também o controlador do Documento, ou seja, o comportamento dinâmico do Documento. Isso significa que a resposta de "o que acontecerá se o usuário selecionar um item de menu referente ao Documento?" é fornecida por métodos dessa classe. Esses métodos que implementam a semântica de um tratador de eventos são derivados da classe *EvtHandler*.

A interface gráfica de um documento é criada na super-classe *Manager*. Na classe *Document*, existem também métodos que referenciam os elementos da interface. Não se pode deixar de ressaltar que, para fins de reutilização, o ideal seria que um documento e suas operações fossem efetivamente independentes de sua interface gráfica.

5.1.2 Estudo de Caso do ET++

O estudo de caso descrito nesta Seção é uma aplicação que possui um relógio com duas Visões (aparências): um relógio digital e um relógio analógico. A aplicação foi desenvolvida usando-se o *framework* ET++. As próximas Seções descrevem a arquitetura e funcionamento da aplicação relógio.

5.1.2.1 Arquitetura da aplicação

A Figura 5.7 mostra a hierarquia de classes da aplicação. O componente Modelo da aplicação corresponde às classes *Manager*, *Application*, *Clock* e *ObjTime*. O componente Visão corresponde às classes *VObject*, *TextItem*, *View* e suas especializações *ClockAnalo*, *ClockDig* e *ClockView*, respectivamente.

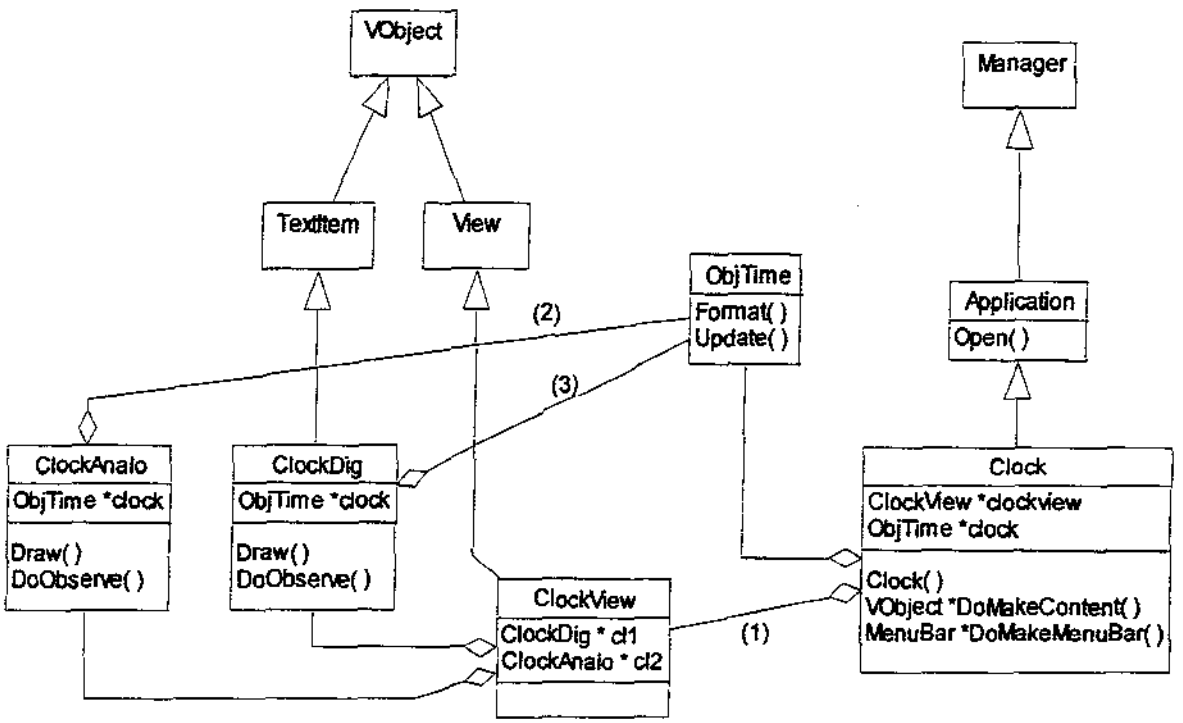


Figura 5.7 : Diagrama parcial de classes da aplicação Relógio

Como dito anteriormente, a camada de interface do sistema, não representada na Figura 5.7, capta os eventos do usuário e os envia para o *FirstHandler* que pode tratá-los ou enviá-los para o Modelo através da cadeia de tratadores de eventos formada pela classe *EvtHandler*. Neste exemplo, o *FirstHandler* é a própria aplicação *Clock*.

Na construção de uma aplicação específica, a classe *Application* deve ser especializada. Para a aplicação relógio, foi definida a classe *Clock* (Figura 5.7) como especialização da classe *Application*, a qual funciona como *Manager* da aplicação. A aplicação *Clock* possui um relógio representado pela classe *ObjTime* do ET++. Um objeto

da classe *ObjTime* possui um *Timer*, objeto responsável pelos eventos síncronos e assíncronos do sistema, a classe *Timer* e sua hierarquia não estão representados na figura, por motivos de espaço. Os métodos *hooks* redefinidos na classe *Clock* são *DoMakeContent()* e *DoMakeMenuBar()*. *DoMakeMenuBar* possui uma implementação *default* na classe *Manager*, neste caso, sua redefinição foi necessária para que nenhuma barra de menu fosse acrescentada à janela. *DoMakeContent()* cria o conteúdo visual da aplicação, um objeto da classe *View* que possui um relógio digital (*ClockDig*) e um relógio analógico (*ClockAnalo*), visões do relógio propriamente dito representado por *ObjTime*.

Observe que existem, nesta implementação, 3 dependências entre Modelo e Visão nas classes específicas da aplicação. Note também que elas são necessárias, pois a aplicação deve conhecer sua visão (relacionamento 1) as visões do relógio precisam recuperar a hora do sistema e portanto conhecer o relógio (relacionamentos 2 e 3).

5.1.2.2 Funcionamento da Aplicação

Esta seção descreve a seqüência de mensagens que implementa as principais operações da aplicação.

O primeiro método a ser executado no funcionamento da aplicação, *Main*, contém um único comando: *return Clock(argc, argv).Run()*. Assim, o construtor da classe *Clock* é chamado, este, por sua vez, chama o construtor da superclasse *Application* que inicia o sistema operacional e de janelas. Em seguida, o método *Run* é chamado para o objeto *Clock*. A seqüência de execução do método *Run* pode ser visualizada na Figura 5.8. Foram omitidas algumas mensagens a fim de simplificar a Figura. Durante a execução do método *Run* são criados o objeto do Modelo e os objetos da Visão. Inicialmente, chama-se o método *Open*, a partir do qual cria-se a janela da aplicação e o seu conteúdo, através do método *hook DoMakeContent* (essas mensagens não são mostradas na Figura) . O método *DoMakeContent*, inicialmente, cria o objeto *clock:ObjTime* (mensagem 2), que inicia um *Timer* e o adiciona ao sistema (mensagens 3 e 4), em seguida, ele chama o construtor de *ClockView* (mensagem 5). O construtor de *ClockView* chama primeiramente o construtor de *ClockDig* que se inclui na lista de observadores do *clock* (*ObjTime*) através do método *AddObserver* (mensagens 6 e 7). Em seguida, chama o construtor de *ClockAnalo*, que, da mesma forma, chama *AddObserver* (mensagem 8 e 9). *ObjTime* é criado com um *timeout* de 1 segundo, isso significa que a mensagem *Update* será enviada para ele a cada 1 segundo. Após o conteúdo da janela ser criado, a janela propriamente dita é então criada, seu *manager* é setado como sendo a aplicação (mensagens 10 e 11). Isso implica que os eventos do usuário, como *cliks* de mouse ou pressionamento do teclado que não são tratados pela janela, deverão ser passados para o próximo *manager* (aplicação). Em seguida, a janela é então aberta através do método *OpenAt* (mensagem 12).

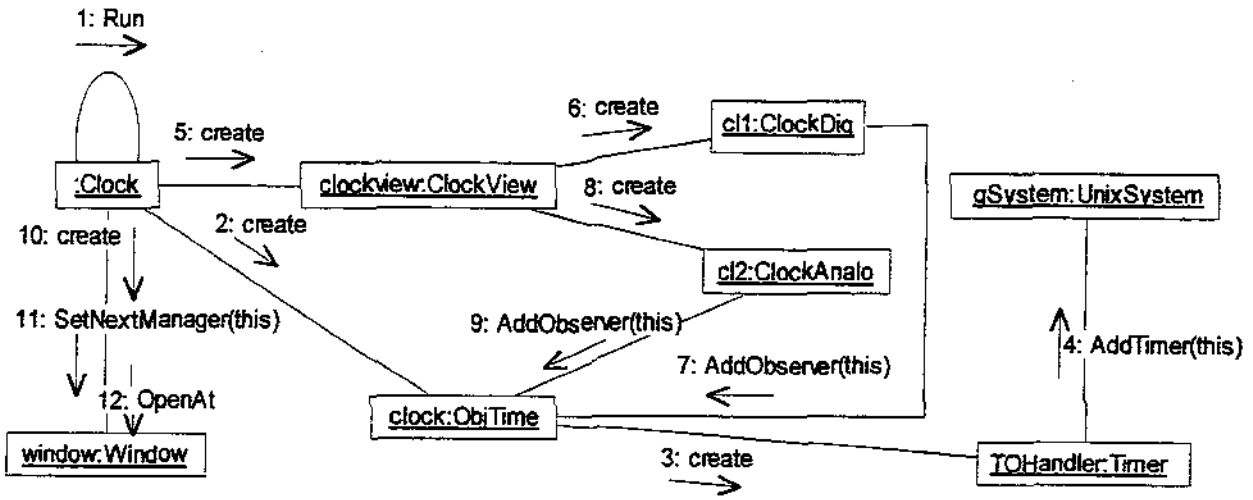


Figura 5.8: Diagrama de colaboração do método *Run* da classe *Clock*

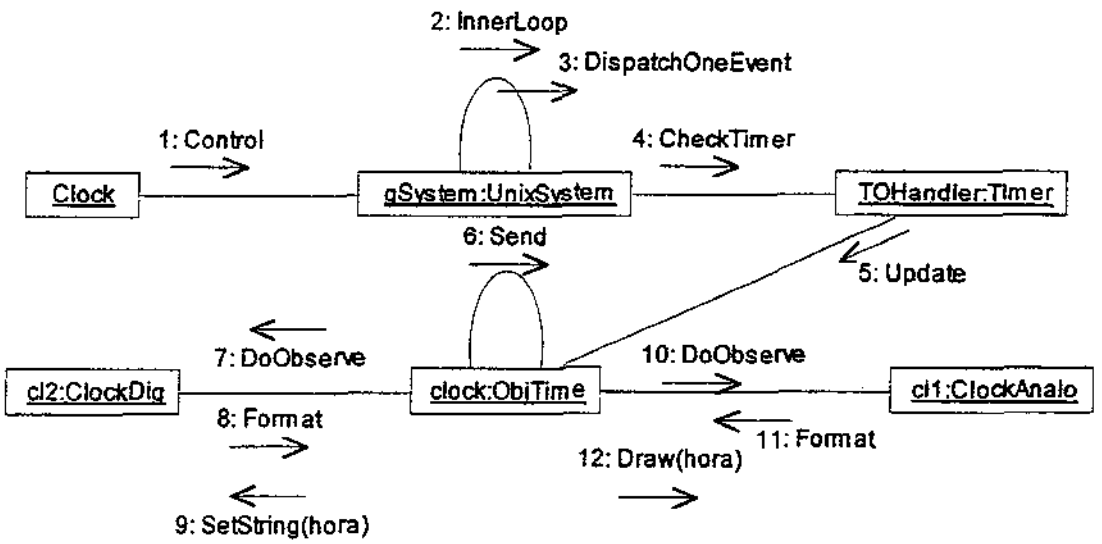


Figura 5.9: Diagrama de colaboração do método *Control* da classe *Clock*

Após a iniciação do sistema e construção da janela, é iniciado, através do método *Control* (Figura 5.9), um laço responsável pelo tratamento dos eventos do sistema e do usuário que só é finalizado quando a aplicação termina. Este laço envia sempre a mensagem *CheckTimer* para o objeto *Timer* (mensagem 4) que, caso tenha se passado 1 segundo, envia a mensagem *Update* para *clock* (mensagem 5). Os métodos *Changed*[11] e *Send* do padrão *Observer* (definidos na classe *Object*) são então chamados (mensagem 6). *Send* chama o método *DoObserve* para o objeto *ClockDig* que recupera a hora exata do sistema, através do método *Format* e seta a sua *string* com a hora atualizada para o relógio digital (mensagens 7, 8, e 9). Em seguida, *Send* chama o método *DoObserve* para o objeto *ClockAnalo* que também recupera a hora exata do sistema, através do método *Format* e redesenha o ponteiro para o relógio analógico (mensagens 10, 11, E 12).

5.2 ET++ Reflexivo

Esta seção descreve o *framework* ET++ reestruturado utilizando reflexão computacional e em seguida mostra a estrutura e funcionamento da aplicação relógio desenvolvida utilizando-se o *framework* reflexivo. O protocolo de metaobjetos de OpenC++ 1.2 [Chi93] foi utilizado na implementação de um protótipo do ET++ Reflexivo construído no Instituto de Computação da Universidade Estadual de Campinas - UNICAMP. A escolha de OpenC++ 1.2 deve-se à sua simplicidade e ao fato de o ET++ ser codificado em C++.

5.2.1 Estrutura

Como dito anteriormente, o ET++ divide o componente Controlador do MVC entre os componentes Modelo e Visão. O componente Controlador implementa parte do protocolo de comunicação entre interface e aplicação, pois o comportamento dinâmico da interface gráfica está diretamente relacionado à funcionalidade da aplicação. O ET++ divide o Controlador da seguinte forma: o recebimento dos eventos do usuário é feito pelo componente Visão. As classes de Interface do Sistema recebem os eventos e os enviam para as classes *Window*, *View* e *VObject*, de acordo com o evento. Já a requisição de serviços do Modelo é feita através das superclasses de Visão e Modelo: *Manager*, *EvtHandler e Object*. Como o objetivo da reestruturação do *framework* é implementar efetivamente a independência de diálogo, decidimos levar para o meta-nível as classes que fazem a conexão entre aplicação e interface, separando assim os serviços administrativos ou não funcionais dos serviços funcionais da aplicação e interface gráfica.

A Figura 5.10 mostra a arquitetura reflexiva do ET++. O *framework* reflexivo contém no nível base os objetos da aplicação (componente Modelo) e os objetos da interface (componente Visão). Esses componentes são independentes e a comunicação entre eles é feita através dos componentes MetaModelo e MetaVisão, implementados no Meta-nível.

[11] O método *Changed* não foi mostrado na figura.

As operações do Controlador ficam divididas entre os componentes MetaVisão e MetaControlador. O MetaModelo implementa as dependências do Modelo em relação à Visão, antes implementadas no ET++ no próprio componente Modelo. Da mesma forma, MetaVisão implementa as dependências da Visão em relação ao Modelo, antes implementadas no ET++ na própria Visão. Quando essas dependências são extraídas para o Meta-nível, obtêm-se componentes (Modelo e Visão) completamente independentes e reutilizáveis.

O funcionamento básico do modelo é o seguinte: qualquer alteração no Modelo é captada pelo MetaModelo, este notifica à MetaVisão que decide quais métodos da Visão devem ser invocados para fazer as alterações correspondentes. Inversamente, as alterações da Visão são captadas pela MetaVisão que as notifica ao MetaModelo o qual chama métodos do Modelo para fazer as alterações correspondentes.

Nas seções seguintes serão mostradas as arquiteturas do MetaModelo e da MetaVisão através da aplicação *Clock* desenvolvida utilizando-se o ET++ reflexivo.

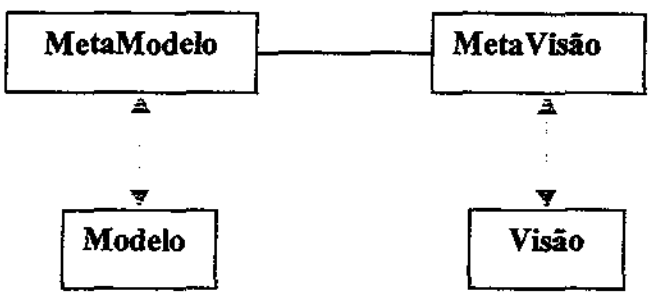


Figura 5.10: Arquitetura do Framework Reflexivo.

5.2.2 Estudo de Caso do ET++ Reflexivo

A mesma aplicação relógio, utilizada no estudo de caso do ET++ (A Seção 5.1.2 foi utilizada no estudo de caso do ET++ Reflexivo. A Figura 5.11 mostra a implementação do nível base da aplicação relógio utilizando-se o ET++ reflexivo. Observe que as classes *Manager, EvtHandler e Object* não se encontram na Figura, pois elas fazem parte do meta-nível nesta abordagem. Observe também que todas as dependências entre Modelo e Visão foram extraídas do nível base e levadas para o meta-nível. Dentre as classes específicas da aplicação, foram eliminadas as dependências (1), (2) e (3) mostradas na Figura 5.7. A aplicação não mais precisa conhecer sua Visão e as visões do relógio não mais precisam conhecê-lo. Dentre as classes do ET++, todas as dependências podem, da mesma forma, ser levadas para o meta-nível. Observe na figura 7 que uma aplicação (Modelo) conhece sua janela e barra de menu (Visão) e a visão (classe *View*) deve também conhecer seu *manager* (aplicação). Essas dependências são implementadas no meta-nível e a conexão com o nível base feita através dos métodos reflexivos *Open* da classe *Manager* e *Update* da classe *ObjTime*. (Utilizamos a notação de OpenC++ versão 1.2 para identificar um método reflexivo.) O método *Open* permite que o controle de execução seja levado para o

metaobjeto que cuidará da construção da janela, barra de menus e outros objetos visuais. Da mesma forma, o método *Update* permite que o controle de execução seja levado para o meta-nível, computações sejam feitas e as alterações nas visões sejam efetuadas. Esses processos são explicitados adiante.

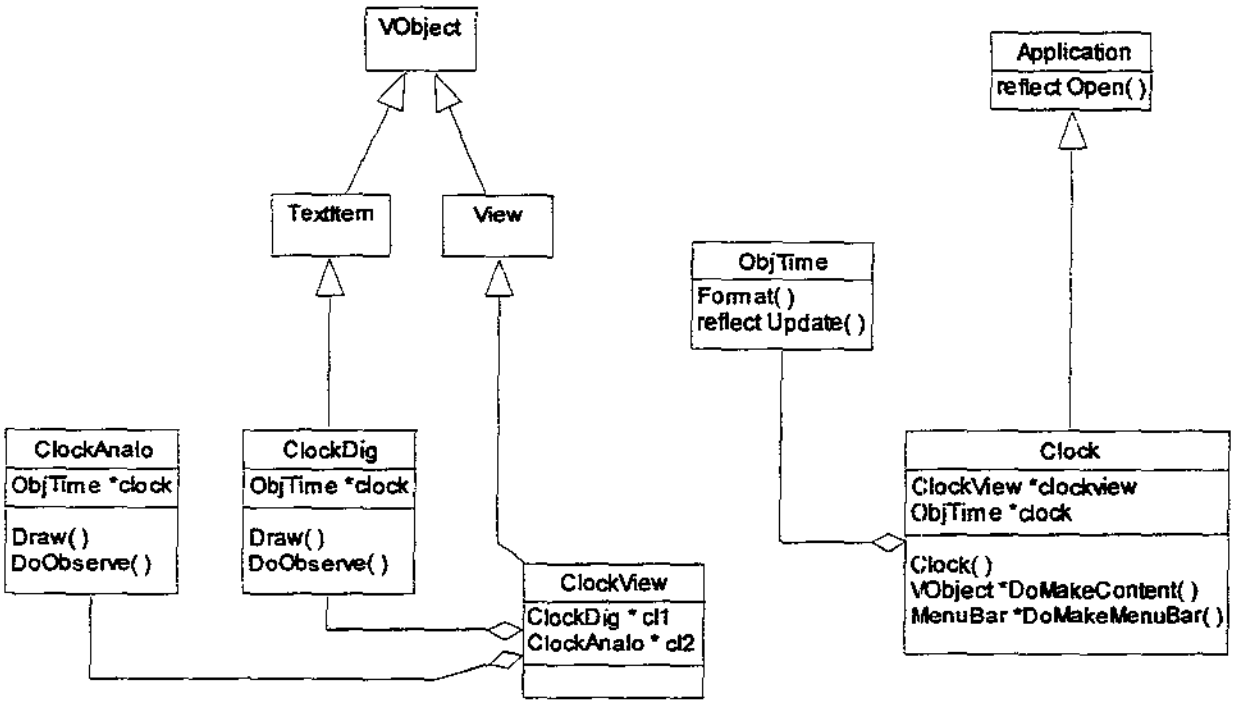


Figura 5.11: Applicação Clock utilizando-se o ET++ Reflexivo.

5.2.2.1 O MetaModelo

A Figura 5.12 mostra o projeto do MetaModelo e seu relacionamento com o Modelo. As classes localizadas acima da linha tracejada pertencem ao meta-nível (MetaModelo) e as localizadas abaixo da linha tracejada pertencem ao nível base (Modelo).

As classes *MetaApplication2* e *MetaDocument2* são derivadas da classe *MetaObject* que implementa o protocolo de metaobjetos em OpenC++ 1.2. Elas são responsáveis pelas interrupções dos métodos reflexivos no nível base e tranferência do controle de execução para o meta-nível, bem como pelo retorno do controle de execução para o nível base. Essas duas classes fazem o papéis físicos de metaobjetos das classes *Application* e *Document*, no entanto os metaobjetos lógicos são as classes *MetaApplication* e *MetaDocument*, respectivamente. Esse projeto ficou bastante modular porque não mistura o protocolo de meta-objetos de OpenC++ com os serviços de gerência e comunicação com a MetaVisão, implementados nos metaobjetos lógicos do MetaModelo.

Observe, na Figura 5.12, que os serviços de gerência da classe *Manager* e os serviços de tratamento de eventos fornecidos pela classe *EvtHandler* foram isolados das classes do Modelo, por meio de reflexão computacional. Essas classes não foram alteradas, nesta

abordagem, elas funcionam segundo os mesmos meta-padrões descritos na seção anterior. A diferença consiste no fato de que o Modelo (*Application* e *Document*) não utiliza seus serviços por meio de herança e sim por meio de reflexão computacional, de forma independente e transparente. Isso faz com que as classes *Application* e *Document* possam ser reutilizadas por interfaces diferentes com esforço mínimo, sem nenhuma alteraçao nos seus códigos. Somente seria necessário que as classes do MetaModelo, dependentes da aplicação, fossem alteradas. Neste exemplo, a única classe do meta-nível dependente da aplicação é a classe *MetaClock*. A descrição das classes é feita a seguir.

Class MetaApplication

Esta classe implementa as operações de comunicação com a interface gráfica e de tratamento de eventos referentes às aplicações de forma geral que, na implementação anterior, faziam parte da classe *Application*. Da mesma forma, as características que não puderam ser implementadas poderão ser acopladas através dos métodos hook. O método *DoMakeContent()* de *Manager*, por exemplo, deve ser redefinido por uma subclasse concreta de *MetaApplication*, também do meta-nível, para implementar a instanciação dos objetos da interface da aplicação.

Class MetaDocument

Semelhantemente à classe *MetaApplication*, esta classe implementa o tratamento de eventos, referente aos documentos de forma geral e os serviços de comunicação com a interface gráfica. Na instanciação de uma aplicação que gerencie documentos, uma classe concreta, subclasse de *MetaDocument*, deve também ser instanciada. Esta subclasse de *MetaDocument* implementará as dependências com a interface específica da aplicação e será a principal tratadora dos eventos da aplicação.

Class MetaClock

A classe *MetaClock* é específica da aplicação, ela é responsável pela instanciação da Visão da aplicação, um objeto da classe *ClockView*. Essa instanciação é feita através do método *DoMakeContent()*. Caso sejam feitas alterações na Visão da aplicação a única classe afetada é a *MetaClock*. A classe *Clock* fica independente de qualquer alteração em objetos da interface.

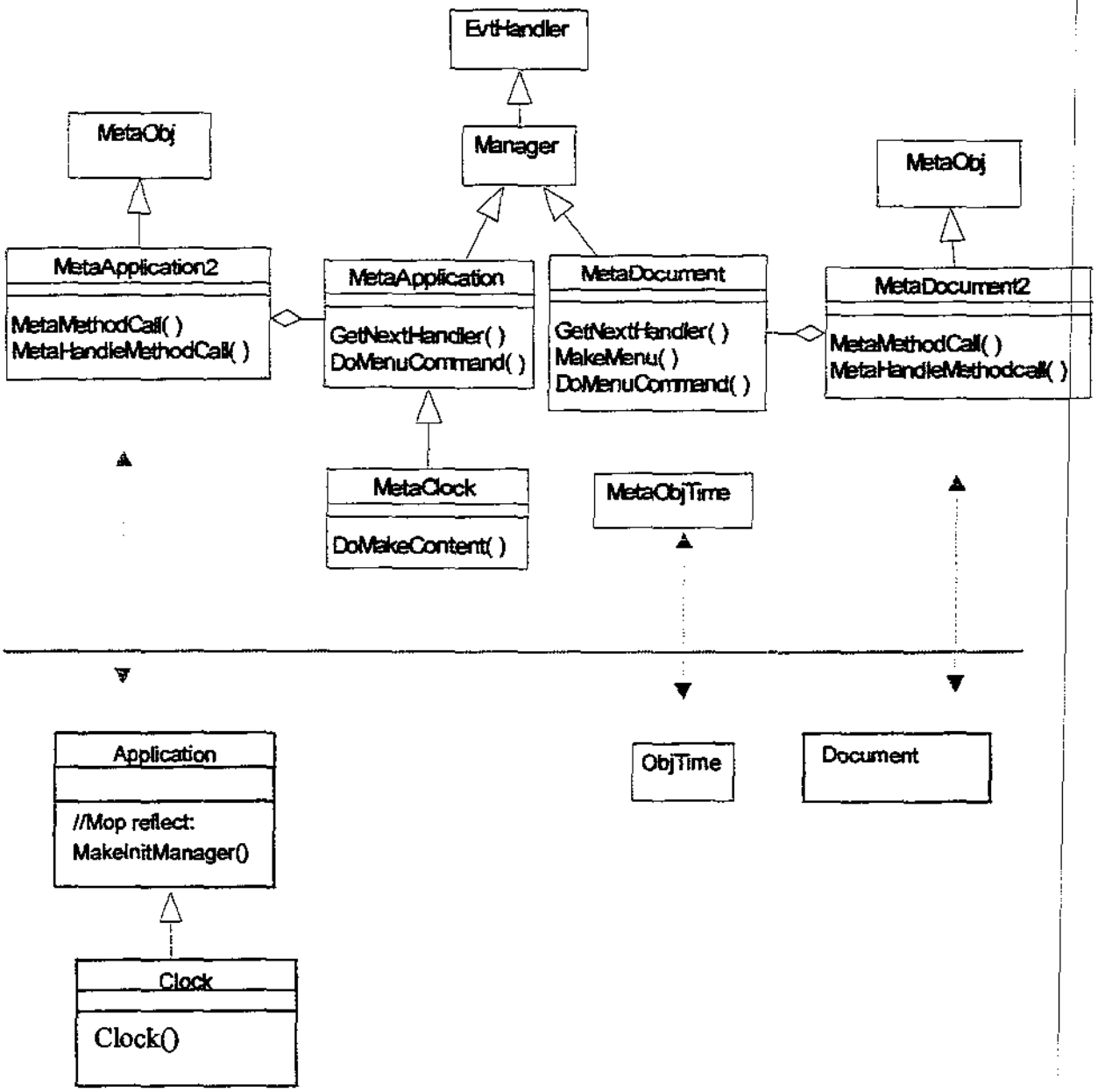


Figura 5.12 : MetaModelo do ET++ Reflexivo.

5.2.2.2 A MetaVisão

A Figura 5.13 mostra o projeto parcial da MetaVisão do ET++ reflexivo. As classes do meta-nível, acima da linha tracejada, são semelhantes às classes do MetaModelo. A classe *MetaVObject2* tem a mesma função das classes *MetaApplication2* e *MetaDocument2*. Ela é a metaclasse física da classe *VObject*. A classe *MetaVObject*, meta-classe lógica de

VObject, implementa o tratamento de eventos nos objetos visuais em geral e as dependências desses objetos em relação aos objetos do modelo, antes implementados na classe *VObject*. Da mesma forma, as classes *MetaMenu*, *MetaView*, e *MetaWindow* implementam os mesmos serviços para os seus objetos logicamente referentes *Menu*, *View* e *Window*, respectivamente. Semelhantemente, as classes do meta-nível específicas da aplicação *Clock MetaClockView*, *MetaClockDig* e *MetaClockAnalo* implementam esses serviços para seus referentes *ClockView*, *ClockDig* e *ClockAnalo* do nível base, respectivamente. O projeto do sistema foi bastante influenciado pelas limitações da linguagem OpenC++1.2 . Em OpenC++ não é permitida a herança de meta-objetos. Assim, os objetos da visão (descendentes de VObject) possuem *MetaVObject2* como meta-objeto físico. Este meta-objeto associa o objeto referente do nível base com o seu meta-objeto lógico correspondente. Assim, as interrupções de execução, feitas por *MetaVObject2*, no nível base, são tratadas pelo meta-objeto lógico do objeto referente em cujo método reflexivo ocorreu a interrupção.

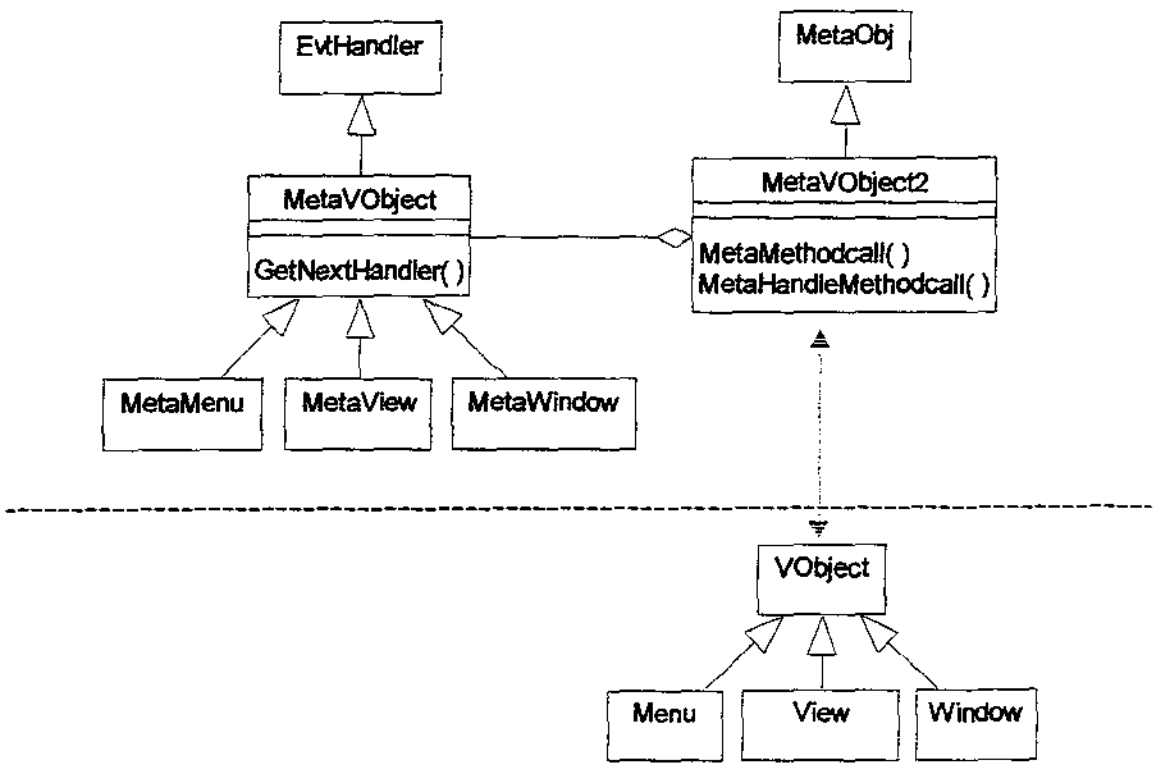


Figura 5.13 : Projeto parcial da MetaVisão.

5.2.2.3 E como ficaram os relacionamentos entre Modelo e Visão?

Nesta abordagem não existem relacionamentos entre Modelo e Visão, estes foram levados para os componentes MetaModelo e MetaVisão como mostra a Figura 5.14. A classe *Manager* possui uma referência para os meta-objetos *MetaWindow* e *MetaMenu* através

dos quais envia mensagens para os objetos *Window* e *Menu* da interface. O método *MakeWindows* da classe *Manager* cria a janela da aplicação. Como dito anteriormente, ele é um método *template* que invoca o método *hook DoMakeContent*. As mensagens da aplicação (*Clock*) são passadas para a Visão (*ClockView*) através de *MetaClock* e *MetaClockView*.

As dependências entre objetos específicos como *ObjTime* e suas visões *ClockDig* e *ClockAnalo*, passam a ser implementadas no meta-nível pelos respectivos metaobjetos, como mostra a Figura 5.15. O padrão Observador continua sendo implementado na classe *Object*, a diferença é que as dependências existem entre os meta-objetos do Modelo e Visão e estes ficam independentes no nível base. O objeto observado passa a ser *MetaObjTime* e os observadores passam a ser *MetaClockDig* e *MetaClockAnalo*. *MetaObjTime* toma o controle de execução quando a hora do sistema é alterada em *ObjTime*, seus observadores são então notificados (*MetaClockDig* e *MetaClockAnalo*), estes fazem as devidas alterações nos seus referentes, as visões *ClockDig* e *ClockAnalo*.

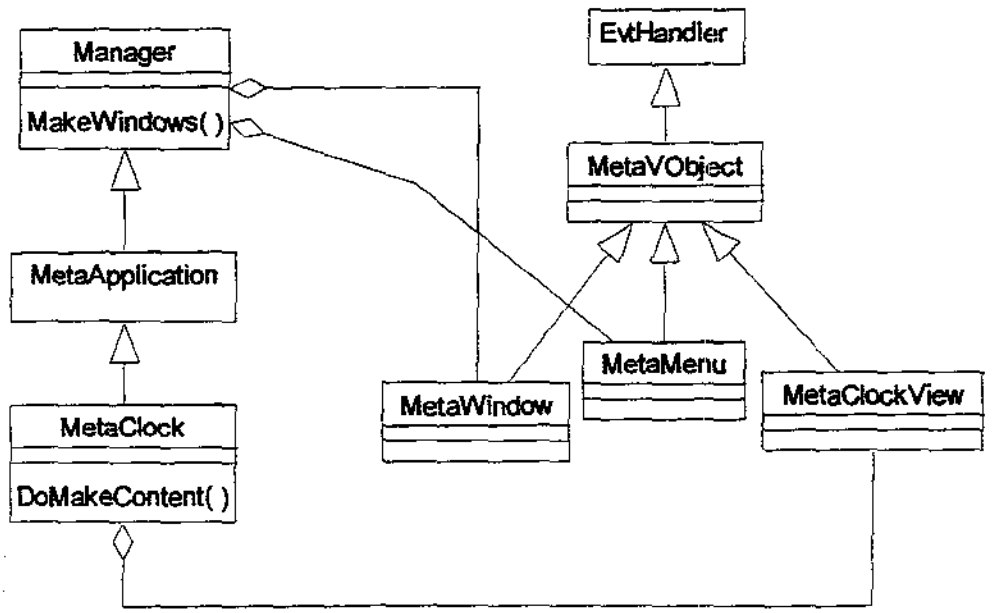


Figura 5.14: MetaModelo e MetaVisão.

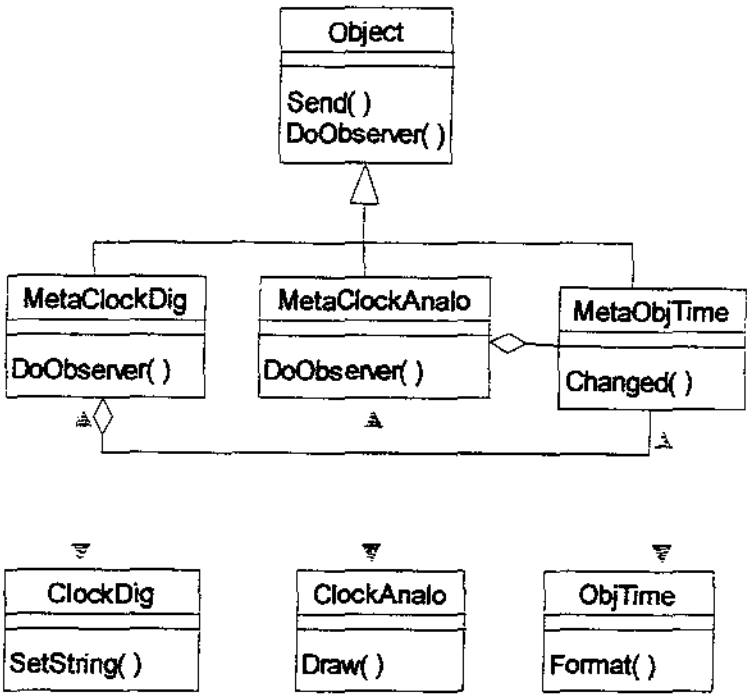


Figura 5.15: Observador Reflexivo.

5.2.2.4 Funcionamento da Aplicação

A seqüência de execução da aplicação é praticamente a mesma descrita na Seção 5.1.5.2. O que a diferencia são as interrupções efetuadas pelo meta-nível para os métodos reflexivos. Inicialmente, *Run* providencia a criação dos objetos *ObjTime* (Modelo) e *MetaObjTime* (MetaModelo), veja Figura 5.16. Em seguida, o método reflexivo *Open* é invocado para o objeto *Clock*, o controle de execução é então tomado por *MetaApplication2* que solicita à *MetaClock* a criação dos objetos visuais da aplicação. Isso é feito da seguinte forma: *MetaClock* inicia a construção da janela da aplicação através de *MakeWindows* (não mostrado na figura). *MakeWindows* chama *DoMakeContent* (também não mostrado na figura), redefinido em *MetaClock*, que cria os objetos *ClockView* e seus objetos visuais *ClockDig* e *ClockAnalo*, juntamente com seus respectivos metaobjetos *MetaClockDig* e *MetaClockAnalo*, como mostra a Figura 5.16. Ao serem criados *MetaClockDig* e *MetaClockAnalo*, incluem-se à lista de observadores de *MetaObjTime*, metaobjeto do relógio (*ObjTime*).

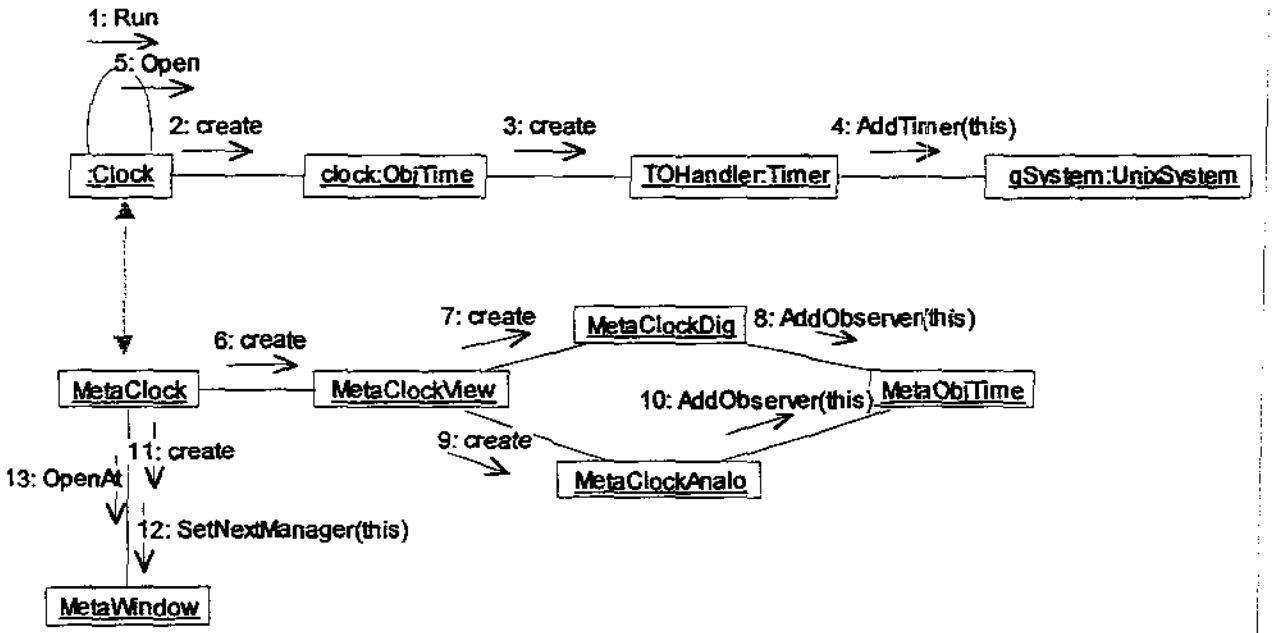


Figura 5.16: Diagrama de colaboração da iniciação da aplicação Relógio na abordagem reflexiva.

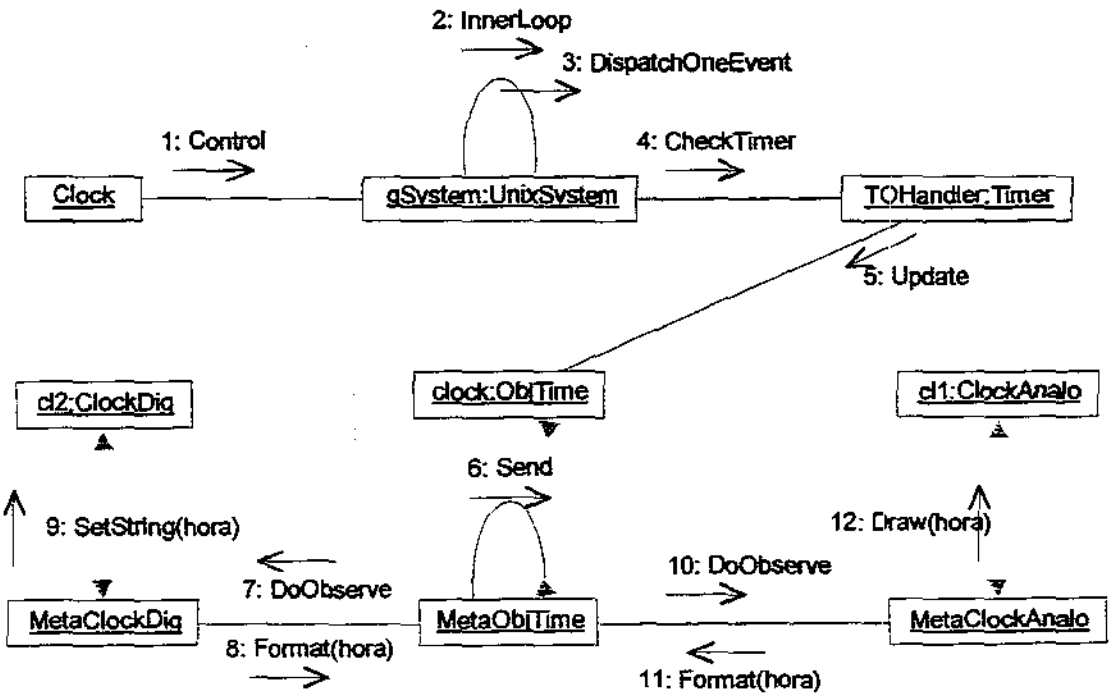


Figura 5.17: Diagrama de colaboração do laço de eventos da aplicação Relógio na abordagem reflexiva.

Após a instanciação dos objetos, o controle de execução volta para o nível base e o método *Control* é chamado. A cada 1 segundo, a mensagem *Update* é passada para *ObjTime*. Como Update é um método reflexivo, sua execução será interrompida por *MetaObjTime* (Figura 5.17) que invocará o método *Send*. *Send* invoca o método *hook DoObserve* para os metaobjetos dos observadores do relógio (*MetaClockDig* e *MetaClockAnalo*). *DoObserver*, redefinido em *MetaClockDig* recupera a hora atual através de *MetaObjTime* e, em seguida, chama *SetString*(hora) de *ClockDig*. Semelhantemente, *DoObserver*, redefinido em *MetaClockAnalo*, recupera a hora, através de *MetaObjTime*, e chama *Draw*(hora). Esse procedimento se repete até a finalização da aplicação.

5.2.2.5 Limitações do ET++ Reflexivo

A construção do protótipo do ET++ Reflexivo foi dificultada por 2 fatores: o tamanho e complexidade do ET++ e as limitações do OpenC++ 1.2, protocolo de meta-objetos utilizado na implementação do ET++ Reflexivo. A complexidade do ET++ e o número de classes dificultaram a separação do núcleo funcional (no nível base) e dos aspectos de controle e comunicação com a interface (no meta-nível). Nós limitamos o número de classes a serem reestruturadas devido a impossibidade de considerarmos todo o *framework*, visto que o ET++ completo possui cerca de 400 classes. Selecionamos para fazer parte do *framework* reflexivo: o *framework* de aplicação e as principais classes da Visão, visto que essas classes implementam o protocolo de comunicação entre Modelo e Visão. Isso ocasionou um trabalho extra de separação das classes que não consideraríamos na reestruturação.

As decisões de implementação do ET++ Reflexivo foram bastante influenciadas pelas limitações do protocolo de meta-objetos de OpenC++ 1.2. A principal limitação deste protocolo é que ele não permite herança de meta-objetos. A Figura 5.18 mostra uma implementação parcial da MetaVisão utilizando herança de meta-objetos. A idéia básica desta implementação é a seguinte: os objetos do nível base implementam suas funcionalidades específicas e os respectivos meta-objetos implementam seus aspectos de controle e comunicação com o Modelo. O fato de espelharmos o projeto do nível base no meta-nível permitiu reutilizar as decisões de projeto do ET++ que implementavam a comunicação entre Visão e Modelo.

Essa implementação não é permitida em OpenC++1.2 pois o pré-processador confunde-se na criação do meta-objeto de *Window*. Em uma hierarquia, todas as classes devem possuir o mesmo meta-objeto para não haver ambigüidades.

A Figura 5.13 mostra a implementação utilizada no ET++ Reflexivo para contornar a impossibilidade de implementar a herança entre metaobjetos. Observe que, nesta implementação, dividimos os meta-objetos em físicos (os que efetivamente realizam interrupções no nível base) e lógicos (que implementam as dependências com o Modelo). A principal conseqüência desta implementação é o aumento do número de classes no meta-nível e uma indireção a mais na comunicação entre Modelo e Visão.

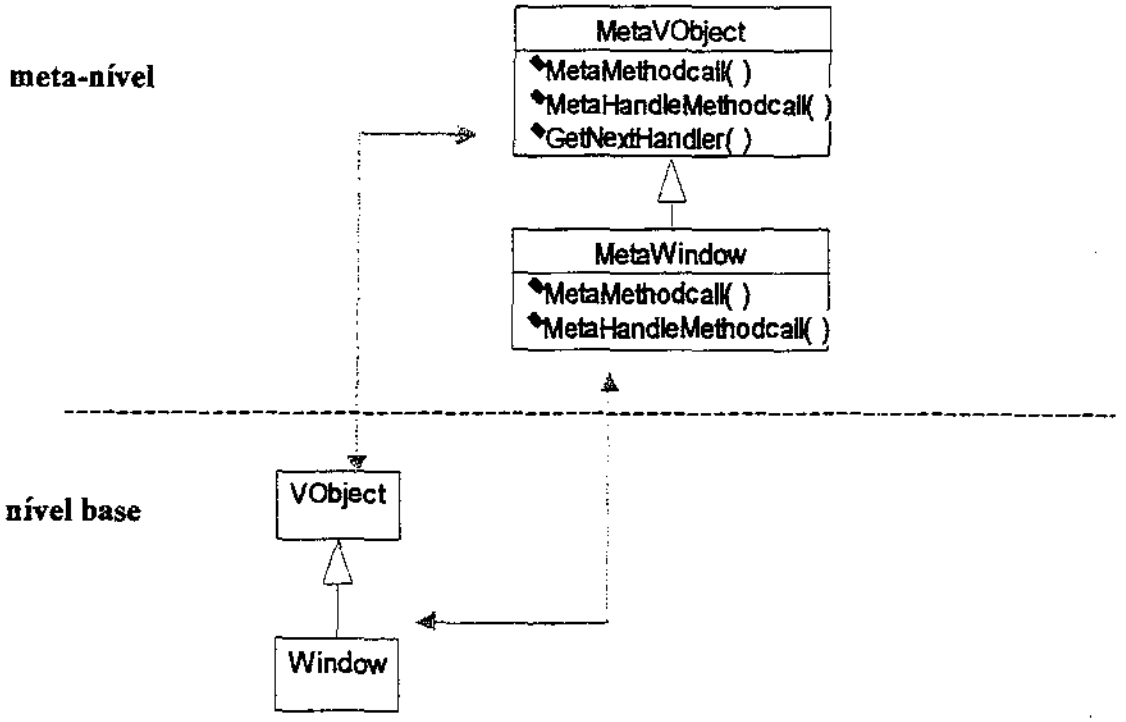


Figura 5.18: Exemplo de herança com metaobjetos.

5.3 Comparação dos Estudos de Caso

Esta Seção faz uma comparação entre os dois estudos de caso, são analisados pontos como independência de diálogo, desempenho e facilidade de uso dos *frameworks* ET++ e ET++ Reflexivo.

5.3.1 Independência de Diálogo

A implementação da aplicação Relógio desenvolvida a partir do ET++ Reflexivo, mostrada na Figura 5.11, possui uma independência de diálogo efetiva enquanto a desenvolvida a partir do ET++, mostrada na Figura 5.7, possui maiores dependências entre Modelo e Visão. Observe, na Figura 5.11, que nas classes da Visão existem somente operações relativas a ela, idem para o Modelo. Diferentemente da implementação da Figura 5.7 em que nas classes da Visão existem chamadas a métodos do Modelo e vice e versa. Utilizando-se o *framework* reflexivo, códigos referentes à comunicação entre Visão e Modelo são implementados em classes separadas, no meta-nível. No nível base, o programador da aplicação apenas identifica os métodos que devem permitir que computações sejam feitas nos outros componentes e os declara como reflexivos.

Os componentes Modelo e Visão do ET++ Reflexivo são completamente independentes podendo ser reutilizados sem grande esforço. As manutenções no *framework* são facilidadas devido à separação de serviços proporcionada pela reflexão computacional. Uma alteração no componente Visão por exemplo, pode afetar no máximo os códigos do meta-nível, de forma alguma tais alterações causam impacto no componente Modelo. A construção de aplicações interativas complexas utilizando o ET++ Reflexivo é mais fácil, pois o projetista pode abordar os aspectos distintos da aplicação separadamente. Ele pode implementar a funcionalidade da aplicação, em seguida, a interface gráfica e finalmente a comunicação entre as duas. A implementação de um componente pode ser feita sem o conhecimento da interface pública do outro componente, exceto a implementação da conexão dos componentes no meta-nível. A efetuação desta exige o conhecimento da interface pública de ambos os componentes. Os projetos dos componentes podem, inclusive, serem executados por projetistas distintos. Na abordagem convencional, mesmo que se tenha um projeto modular, deve existir interação entre o projetista da interface e o do Modelo pois as dependências são implementadas dentro dos componentes.

5.3.2 Desempenho

A desvantagem do ET++ Reflexivo em relação ao ET++ convencional é uma possível perda de eficiência causada pelas interrupções do meta-nível e as indireções dos meta-objetos. Uma requisição de serviços da interface gráfica para a aplicação e vice e versa, no ET++, é feita diretamente. Na abordagem reflexiva, essas requisições passam por uma interrupção e dois metaobjetos, no meta-nível, antes de chegarem aos seus destinos. Deve ser ressaltado também que o uso de um protocolo de metaobjetos mais efetivo reduziria o número de indireções da implementação, pois uma requisição feita da interface para a aplicação e vice e versa poderia passar apenas por uma interrupção e um meta-objeto no meta-nível. As limitações de OpenC++1.2 nos impossibilitou de implementar o *framework* reflexivo de forma mais eficiente, como mostrou a Seção 5.2.2.5.

Nenhuma medida de desempenho foi feita na realização do experimento, entretanto consideramos importante a coleta de medidas dos *overheads* causados pelo uso de reflexão computacional em sistemas interativos. Esta análise de desempenho é considerada um trabalho futuro.

5.3.3 Uso do Framework ET++ Reflexivo e ET++

Esta Seção discute a complexidade de desenvolvimento de uma aplicação utilizando o *framework* ET++ reflexivo se comparado ao uso do ET++. A Seção 5.3.1 comparou a independência de diálogo obtida nos dois estudos de caso e verificou, baseada nos experimentos das Seções 5.1 e 5.2, que a independência obtida com a abordagem reflexiva traz importantes conseqüências para o *framework* reflexivo. Uma dessas conseqüências é

um maior suporte na construção de aplicações complexas devido a facilidade de abordar aspectos distintos da aplicação separadamente, devido a independência de aspectos funcionais e não funcionais proporcionada pela reflexão computacional. Por outro lado, o uso do *framework* ET++ Reflexivo exige um conhecimento prévio do protocolo de meta-objetos OpenC++1.2. Dessa forma, qualquer analista, que não conheça reflexão computacional e portanto nenhuma linguagem reflexiva, dirá que o desenvolvimento de uma aplicação a partir do ET++ Reflexivo é mais complexo que o desenvolvimento de uma aplicação a partir do ET++. Isso se deve ao não conhecimento de um nova técnica de programação: reflexão computacional. Para fins de comparação dos procedimentos para desenvolvimento de aplicações utilizando os *frameworks*, pode-se enumerar os passos, em alto nível, para desenvolvimento de uma aplicação a partir do ET++ Reflexivo:

1) Identificação dos objetos do Modelo que implementarão a funcionalidade da aplicação – o usuário do *framework* deve verificar quais objetos podem ser herdados do *framework* e deve implementar os pontos flexíveis do *framework* para adaptação das características específicas de sua aplicação.
2) Identificação dos objetos da Visão que implementarão os objetos gráficos da Interface Gráfica da aplicação – da mesma forma o usuário deve reutilizar o projeto do *framework* e implementar a parte específica de sua aplicação.
3) Identificação dos objetos do Modelo (Visão) que deverão interagir com a Visão (Modelo).
4) Implementação dos meta-objetos dos objetos identificados em 3, criando-se assim o MetaModelo e a MetaVisão – os objetos identificados em 3) que fazem parte do *framework* já possuem meta-objetos implementados. O usuário do *framework* deve então verificar quais meta-objetos específicos ele precisa criar e herdar dos meta-objetos implementados no *framework* reflexivo.
5) Identificação dos métodos do Modelo que causam alteração na Visão e vice e versa para implementá-los como reflexivos, a fim de que eles transfiram o controle de execução do nível base para o meta-nível.
6) Implementação das dependências entre o MetaModelo e a MetaVisão – grande parte dessas dependências já foram implementadas no meta-nível do *framework*, o usuário deve implementar as dependências específicas de sua aplicação.

O desenvolvimento de uma aplicação utilizando a abordagem convencional do ET++ envolveria os passos 1, 2, 3 e 6, sendo que no passo 6 as dependências seriam implementadas diretamente nos objetos identificados em 3 (que não fariam parte do *framework*). Os passos 4 e 5 são os passos que envolvem a construção do meta-nível e a localização dos pontos no código do nível base onde ocorrerão as interrupções do meta-nível. Estes passos são considerados simples, se o desenvolvedor conhece o protocolo de meta-objetos a ser utilizado. É um custo a mais no desenvolvimento que é recompensado com os benefícios citados na Seção 5.3.1. Tal custo no desenvolvimento torna-se

insignificante quando consideramos a facilidade de manutenção obtida com a arquitetura da aplicação reflexiva.

Vale ressaltar que o desenvolvimento de ferramentas CASE apropriadas para desenvolvimento de aplicações que utilizem reflexão computacional pode facilitar o trabalho de codificação dos objetos do meta-nível. Foi visto anteriormente, nas Seções que discutem as limitações das conseqüências de uso dos padrões MVC e MVCR (Seções 3.1.1.4 e 4.4.2), que existe uma certa dificuldade na utilização de padrões de arquitetura com ferramentas modernas de construção de interface do usuário. Isso acontece porque tais ferramentas não geram o código da interface baseado na arquitetura do padrão. Utilizar o padrão, neste caso, implica na adaptação do código gerado à arquitetura do padrão, e esse é um trabalho árduo. Entretanto, os padrões de arquitetura seriam facilmente utilizados nas aplicações se existissem *frameworks* baseados nos padrões que possuíssem ferramentas próprias de apoio ao desenvolvimento de aplicações. O *framework* MFC, por exemplo, possui ferramentas de apoio ao desenvolvimento de aplicações e o ET++ possui um próprio ambiente de programação que facilita o uso do *framework*. Semelhantemente, para facilitar o uso do ET++ Reflexivo, poder-se-ia utilizar o ambiente de programação do ET++ para desenvolvimento do nível base da aplicação e uma ferramenta apropriada deveria ser construída e integrada a esse ambiente para facilitar a construção do meta-nível da aplicação.

5.4 Resumo

Os estudos de caso utilizando o ET++ Reflexivo e o ET++ possibilitou uma comparação prática dos padrões de arquitetura MVR e MV. Conforme visto na seção 4.5 as semelhanças entre o MV e o MVC são grandes e no aspecto independência de diálogo, que se refere à comunicação entre Modelo e Visão, eles são equivalentes, portanto as conclusões do experimento podem ser estendidas para uma comparação entre o MVC e o MVCR. Na essência, a comparação que se faria entre o MVC e o MVCR é a mesma, mudariam pequenos detalhes devido a existência de um componente a mais no nível base (Controlador) e o seu respectivo componente no meta-nível (MetaControlador). Todas as considerações que foram feitas para os componentes Modelo e Visão seriam também feitas para o componente Controlador. Se o ET++ fosse baseado na íntegra no MVC, o ET++ Reflexivo seria baseado no MVCR, isso implicaria em uma independência ainda maior entre as funcionalidades dos objetos e o desempenho seria inferior ao desempenho do ET++ Reflexivo baseado no MVR.

Experimentos práticos com padrões de arquitetura exigem o uso de softwares construídos baseados nos padrões de arquitetura. Escolhemos trabalhar com o ET++ pelo fato de ser um *framework* bem construído baseado em uma simplificação do MVC. Assim, precisaríamos construir o *framework* baseado da mesma forma em uma variação do

MVCR. Para facilitar a comparação dos padrões achamos apropriado reestruturar o *mesmo framework* ET++ utilizando o padrão MVCR. O *framework* também poderia ser construído do zero baseado no padrão de arquitetura MVCR, não adotamos esta prática devido a impossibilidade de construirmos um *framework* para posteriormente efetuar os experimentos em tempo hábil para realização deste trabalho.

Capítulo 6

Conclusões e Trabalhos Futuros

Este trabalho propôs um padrão de arquitetura para a construção de *frameworks* para interfaces homem-computador - O MVCR (Modelo Visão Controlador Reflexivo). O MVCR é uma combinação do padrão de arquitetura MVC (Modelo Visão Controlador) e o padrão de arquitetura Reflexão Computacional. As principais características do MVCR são (i) a divisão de um sistema interativo em três componentes bem definidos como no MVC e (ii) a separação de propriedades funcionais e não funcionais proporcionada pelo padrão de arquitetura Reflexão computacional. Essas características proporcionam uma independência de diálogo efetiva aos *frameworks* para interfaces homem-computador facilitando o desenvolvimento, manutenção e reutilização dos componentes de uma aplicação interativa.

O desenvolvimento de uma aplicação interativa é facilitado quando a implementação do núcleo funcional pode ser efetuada independentemente da interface gráfica conforme o conceito de independência de diálogo [HHi89]. Isso acontece devido à separação de serviços proporcionada pelo padrão. As dependências entre aplicação e interface são implementadas no meta-nível separadamente da funcionalidade dos mesmos. A manutenção desses componentes também é facilitada devido ao mesmo motivo. Uma alteração na funcionalidade do componente Modelo que não afete a apresentação dos objetos, por exemplo, não atinge o componente Visão, devido ao isolamento dos aspectos de controle que implementam a comunicação com a Visão. Manutenções no componente Visão também não são propagadas para o componente Modelo, pelo mesmo motivo. A reutilização dos componentes é facilitada devido à independência existente entre eles. Uma interface gráfica poderia ser acrescentada a uma aplicação já implementada nesta abordagem sem grande esforço, pois as dependências com a interface não alterariam o código fonte da aplicação. Bastaria identificar na aplicação (Modelo) os objetos e métodos que se comunicam com a interface, espelhá-los no meta-nível (MetaModelo) e implementar a conexão com a MetaVisão e a Visão propriamente dita.

A limitação do MVCR seria uma possível perda de eficiência, caso o número de interrupções efetuadas pelo meta-nível seja muito grande, pois, nesta abordagem, a requisição de serviços de um componente para o outro não é feita diretamente; passa antes pelo meta-nível. O número de classes a serem projetadas aumentam na abordagem reflexiva, pois, se no MVC um objeto implementa sua funcionalidade e as dependências

com objetos de outros componentes, no MVCR existirá um objeto implementando a funcionalidade e um metaobjeto implementando a dependência.

O capítulo 2 descreveu conceitos básicos relacionados ao modelo de objetos. O capítulo 3 descreveu *frameworks* para interfaces homem-computador (*GUI frameworks*). O capítulo 4 apresentou o padrão de arquitetura MVCR. As vantagens e limitações do MVCR foram avaliadas baseando-se em um estudo comparativo de uma aplicação prática utilizando o *framework* ET++, descrito no capítulo 5. Finalmente este capítulo conclui este trabalho, apresentando as contribuições, as dificuldades encontradas, e os trabalhos futuros a serem desenvolvidos.

6.1 Contribuições

As principais contribuições deste trabalho são:

- A proposta de um padrão de arquitetura para a construção de *GUI frameworks*, baseado nos padrões de arquitetura MVC (*Model-View-Controller*) e Reflexão Computacional.
- Uma implementação do padrão de projeto Observador utilizando Reflexão computacional, originando o chamado Observador Reflexivo, que pode ser utilizada para implementar dependência entre objetos, de forma que as dependências sejam implementadas no meta-nível, independentemente da funcionalidade do objeto.
- A aplicação do padrão de arquitetura MVR, uma simplificação do MVCR, na reestruturação do projeto do *framework* ET++, obtendo-se o ET++ Reflexivo.
- Implementação de um protótipo do ET++ Reflexivo.
- Um estudo comparativo entre a abordagem convencional do MVC e a abordagem reflexiva do MVCR, através da implementação de uma mesma aplicação a partir do ET++ e a partir do ET++ Reflexivo.
- Uma solução para o problema da implementação de herança entre metaobjetos em OpenC++ 1.2. Implementamos a idéia de meta-objeto físico e meta-objeto lógico. O meta-objeto físico é descendente da classe *MetaObject* do protocolo de OpenC++1.2 que realiza as interrupções no nível base, entretanto, ele delega para o meta-objeto lógico o tratamento da interrupção. Assim, pôde-se espelhar os objetos do nível base e seus relacionamentos no meta-nível em forma de meta-objetos lógicos.

6.2 Dificuldades Encontradas

As principais dificuldades na realização deste trabalho estão relacionadas ao experimento prático descrito no capítulo 5. Mais precisamente, estão relacionadas à reestruturação do *GUI framework* ET++ utilizando o padrão de arquitetura MVCR e à implementação do

ET++ Reflexivo utilizando o protocolo de metaobjetos OpenC++1.2. Assim, podemos enumerar duas principais dificuldades encontradas:

- O tamanho e complexidade do ET++. O ET++ é um *framework* bastante completo possuindo cerca de 400 classes. Na construção do protótipo, tivemos de limitar o número de classes devido à inviabilidade de reestruturar todo o *framework*. Assim, reestruturamos as classes do *framework* de aplicação, as classes do Modelo e parte das classes da interface gráfica. O ET++ implementa todos os recursos gráficos que uma aplicação interativa precisa. O protótipo do ET++ Reflexivo possui apenas os recursos básicos como janelas, menus e barras de rolagem, recursos suficientes para implementar uma aplicação simples como a descrita (aplicação Relógio) e avaliar as vantagens e limitação do padrão de arquitetura proposto. A separação das classes da interface que não iríamos incluir no ET++ Reflexivo não foi simples também, pois teríamos de garantir que elas não seriam referenciadas pelas classes do protótipo.
- Limitações de OpenC++ 1.2. O protocolo de metaobjetos OpenC++ 1.2 foi preferido ao OpenC++ 2.0 devido à facilidade de uso e também pelo fato de existirem implementações conhecidas neste protocolo. Este protocolo atendeu às necessidades mas influenciou no projeto do protótipo, pois ele não permite herança de meta-objetos. A solução para este problema aumentou o número de classes no meta-nível reduzindo a eficiência do ET++ Reflexivo.

6.3 Trabalhos Futuros

Durante a realização deste trabalho foram identificados os seguintes trabalhos futuros:

- Extensão do *framework* para o desenvolvimento de aplicações gráficas distribuídas. Neste caso, além das dependências entre os componentes Modelo, Visão e Controlador, os serviços relativos à distribuição também poderão ser implementados no meta-nível. Neste caso, alguns experimentos práticos na literatura [CM93] mostram que o *overhead* associado às interrupções efetuadas pelo meta-nível é insignificante quando comparado ao tempo de latência da rede.
- Uma coleta de medidas dos *overheads* causados pelo uso de reflexão computacional em sistemas interativos. A comparação em tempo real do desempenho do MVC e MVCR.
- Desenvolvimento de um *Cookbook* para uso do padrão MVCR na construção de sistemas interativos.
- A implementação de novos experimentos usando um protocolo de meta-objetos mais efetivo como por exemplo o Guaraná [OGB98], descrito na Seção 2.6.5.5.

Bibliografia

[AGG96] M. Ancona, G. Godero, V. Gianuzzi, A. Clematis e M. L. Lisboa. *Reflective arquitectures for reusable falt-tolerant software.* XXI Conferência Latino-Americana de Informática, Março 1996.

[AIS77] C. Alexander, S. Ishikawa e M. Silverstein. A Pattern Language. *Oxford University Press.* 1977.

[Ale79] C. Alexander. The Timeless Way of Building, *Oxford University Press*, 1979.

[BFM92] Dennis J. M. J. de Baar, James D. Foley, Kevin e E. Mullet. Coupling Aplication Design and User Interface Design. *Proceedings of Human Factors in Computing Systems- CHI'92*, Maio 1992.

[BKK86] D. G. Bobrow, G. Kiczales, K. Kahn, L. Masinter, M. Stefik e F. Zdybel. Common-Loops: Merging Lisp and Object Oriented Programming. *OOPSLA'86- Conference Proceedings. Sigplan Notices,* 21(11), Novembro, 1986.

[BMR96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad e Michael Stal. *Pattern-Oriented Software Arquitecture, A System of Patterns*. John Wiley & Sons Ltd. Inglaterra, 1996.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications, Second Edition.* The Benjamim/Cummings Publishing Company. Rational, Santa Clara, Califórnia 1994.

[BRJ97] Grady Booch, James Rumbaugh e Ivar Jacobson. Unified Modeling Language - Version 1.01. *Rational Software Corporation, 1997.*

[Chi93] Shigeru Chiba. Open C++ Programer's Guide. *Department of Infomation Science-University of Tokyo* 93-3, 1993.

[Chi96] Shigeru Chiba. Open C++ Programer's Guide for Version 2. *Xerox PARC and University of Tokyo*, 1996.

[CM93] Shigeru Chiba e Takashi Masuda. Designing an Extensible Distributed Language

with a Meta-Level Arquitecture. *ECOOP'93,* 707 Julho 1993.

[Coa92] Peter Coad. Object Oriented Patterns. *Comunication of the ACM.* 35(9), Setembro, 1992.

[Cop92] J. O Coplien. *Advanced C++ - Programming Styles and Idioms.* Adson Wesley, Reading, MA, 1992.

[Cou87] J. Coutaz. PAC, an Object-Oriented Model for Dialog Design, Human-Computer Interaction. *INTERACT'87 proceedings,* pp 431-436, Stuttgart, Germany, 1987.

[CPr95] Marcelo Campo e Roberto Tom Price. O uso de Técnicas Visuais e Navegacionais para a compreensão de Frameworks Orientados a Objetos. *IX Simpósio Brasileiro de Engenharia de Software,* Recife - Brasil, Outubro, 1995.

[CPr96] Marcelo Campo e Roberto Tom Price. Um Ambiente para Sonorização Não Intrusiva de Aplicações Orientadas a Objetos. *X Simpósio Brasileiro de Engenharia de Software.* São Carlos - Brasil, Outubro, 1996.

[CRB97] Marília G. Coelho, Cecília M. F. Rubira e Luiz E. Buzato. Uma Abordagem Reflexiva para construção de Frameworks para Interfaces Homem-Computador. *Simpósio Brasileiro de Engenharia de Software. Fortaleza-CE Brasil,* Outubro, 1997.

[Elt95] Elton José da Silva. Representação em Projeto de Interfaces Homem-Computador: Estudo, Aplicação e Propostas de Extensão do Formalismo UAN. Faculdade de Engenharia Elétrica - UNICAMP, Julho 1995.

[Fer89] Jacques Ferber. Computational Reflection in Class Based Object Oriented Languages. *OOPSLA'89 Proceedings,* Outubro 1989.

[FHL97] Froehlich G., Hoover H., Liu L., Sorenson P., Hooking into Object-Oriented Application Frameworks. *In: Proc. ICSE'97, International Conference on Software Engineering,* Boston, May 1997, 491-501.

[Fir93] Donald G. Firesmith. Frameworks: The Golden Path to Object Nirvana. *JOOP, 6(6)* Outubro, 1993.

[FNP95] Jean-Charles Fabre, Vincent Nicomette, Tanguy Pérenon, Zhixui Wu e Robert Stroud. *Implementing Fault Tolerant Aplications using Reflective Object Oriented Programing.* 25° Symposium on Fault-Tolerant Computing Systems - FTCS-25,

Pasadena - CA , Junho 1995.

[GB94] Grady Booch. Designing an Application Framework. *Dr. Dobb's Journal,* 19(2), Fevereiro, 1994.

[GHJ95] Erich Gama, Richard Helm, Ralph Johnson e John Vlissides. Design Patterns: *Elements of Reusable Object-Oriented Software.* Addison Weley Publishing Company, Abril 1995.

[GM95] D. Gangopadhyay e S. Mitra. Understanding Frameworks by Exploration of Exemplars. *In Proceedings of 7th International Workshop on Computer Aided Software Engineering (CASE -95),* Toronto Canada, 1995, pp 90-99.

[Gol97] M. Golm. Design em implementation of a meta Arquitecture for Java. Diplomarbeit im Fach Informatik, Friedrich-Alexander Universitat, Erlanger-Numberg. Janeiro de 1997.

[Gold83] Adele Goldberg. Smalltalk-80 - The Interactive Programming Enviroment. *Addison Wesley,* 1983.

[HHa93] Deborah Hix e H. Rex Hartson. Developing User Interfaces: ensuring usability through product and process. *John Willey and Sons,* 1993.

[HHi89] H. Rex Hartson and Deborah Hix. Human-Computer Interface Development: Concepts. and Systems for its Manegement. *ACM Computing Surveys,* 21(21):5-92, Março 1989.

[Hmo90] Volker Harslev e Ralf Moller. A Framework for Visualizing Object Oriented Systems. *Proceedings of the 4° European Conference on OOP - ACM Sigplan Notices,* 25(10), Outubro, 1990

[Jaa95] Ari Jaaksi. Object Oriented Specification of User Interfaces. *Software Practice And Experience.* Novembro, 1995.

[Jac92] I. Jacobson, et. al. *Object-Oriented Software Engineering, a Use Case Driven Appoach,* Addison-Wesley, Reading, MA, 1992.

[JFo88] R. E. Johnson e B. Foote. Designing Reusable Classes. *Journal of Object Oriented Programing,* Junho, 1988.

[Joh92] Ralph E. Johnson. Documenting Frameworks using Patterns. *Proceedings OOPSLA*

'92, *ACM SIGPLAN Notices*, vol. 27, nº 10, Oct. 1992, pp. 63-76.

[KJ94] Kent Beck e Ralph Johnson, Patterns Generate Architectures, *Proceedings ECOOP'94*, M. Tokoro, R. Pareschi (Ed.), LNCS 821, Springer-Verlag, Bologna, Italy, July 1994, pp. 139-149.

[KP88] G.E.Krasner e S. T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object Oriented Programming*. 1 (3), August-September 1988, pp 26-49.

[KZ90] Sasa Konecni e Isabelle de Zegher. Object-Oriented Programming and the design of an Open System for User Interfaces. Bélgica. Abril 1990.

[LC96] Maria Lúcia Lisboa e Gerson G. Hourich Cavalheiro. Reflexão Computacional sobre técnicas de Tolerância a Falhas. *VI Simpósio de Computadores Tolerantes a Falhas*. RS - Brasil, Março, 1996.

[Lew95] Ted Lewis, Larry Rosenstein, Wolfgan Pree at al. *Object Oriented Application Frameworks*. Manniing Publications Co. Greenwich, 1995.

[LM97] F. Losavio e A. Matteo. A Method for User-Interface Development. *JOOP – The Journal of Object-Oriented Programming*. 10(5). Setembro.

[LRB96] Maria Lúcia Blanck Lisbôa , Cecília M. Fischer Rubira e Luiz E. Buzato. Arquitetura Reflexiva para o Desenvolvimento Software Tolerante a Falhas. *XXIII Seminário Integrado de Software e Hardware- SEMISH XXIII*, Recife, PE, Agosto, 1996.

[LRu96] Maria Lúcia Blanck Lisbôa e Cecília M. Fischer Rubira. Técnicas de orientação a objetos para tolerância a falhas. *I Simpósio Brasileiro de Linguagens de Programação- I SBLP*, Belo Horizonte, MG, Setembro, 1996.

[LTM93] Paulo H. C. Lisboa, José F. Tepedino e Silvio L.Meira. Reflexão Computacional em Smalltalk. *Revista Brasileira de Computação*, 7(1) Jul/Dez 1993.

[Mae87] Pattie Maes. Concepts and Experiments in Computacional Reflection. *ACM SIGPLAN Notices, OOPSLA'87*, 22(12):147-155, Dezembro 1987.

[McA95] Jeff McAffer. Meta-Level Programing with CoadA. *ECOOP'95 - 9th European Conference- LNCS*, 952, Dinamarca, Agosto, 1995.

[Mey88] Ware Meyers. Interview with Wilma Osborne. *IEEE Software* 5(3): 104-105, 1988.

[MWY91] Satoshi Matsuoka, Takuo Watanabe e Akinori Yonezawa. Object-Oriented Concurrent Reflective Arquitectures. Object-Based Concurrent Computing. *ECOOP'91 Workshops*, Julho 1991.

[Orf96] H. E. Orfali. *The Essential Distributed Objects Survival Guide*. Wiley, 1996.

[OGB98] A. Oliva, I. C. Garcia e L. E. Buzato. The Reflective arquitecture of Guaraná. Techinical Report IC-98-14, Instituto de Computação, Universidade Estadual de Campinas, Abril, 1998.

[Pae88] Andreas Paepcke. PCLOS: A Flexible Implementation of CLOS Persistence. E*COOP'88*, 322, Oslo Agosto, 1988.

[Pree94] Wolfgang Pree. *Meta patterns - a means for capturing the essentials of reusable object-oriented design.* Springer- Verlag, 1994.

[Rao91 Ramana Rao. Implementation Reflection in Silica. *ECOOP'91 Lecture Notes in CS- 512* Geneva, Switzerland, julho 1991.

[Rumb91] James Rumbaugh, Michael Blaha, William Premerlani et al. *Object-Oriented Modelling and Design*, Prentice Hall. 1991.

[Sch96] Douglas Schmidt em entrevista a Alda Campos. Design Patterns: o simples está na moda. *COMPUTERWORLD*. Vol III, nº 153. Março, 1996.

[Str92] Robert Stroud. Transparence and Reflection in Distributed Systems. *ACM SIGOPS European Workshop*, Abril 1992.

[SW88] R. J. Stroud and Z. Wu. Using Meta-Objects to Adapt a Persistent Object System to Meet Aplication Needs. *University of Newcasttle*, 1988.

[SW95] Robert Stroud e Z. Wu. Using MetaObject Protocols to Implement Atomic Data Types. *ECOOP'95- LNCS. 952,* Dinamarca, Agosto 1995.

[TT89] Yasuhiko Yokote Fumio Teraoka e Mario Tokoro. A Reflective Arquitecture for Object-Oriented Distributed Operating System. *ECOOP'89 -Proceedings of the Third European Conference on Object-Oriented Programing,* 1989.

[UCM96] Kazutomo Ushijima, Shigeru Chiba e Takashi Masuda. Meta-Level Programming for Simplifying Library Protocols. *Submitted to ISOTAS'96*, Xerox Parc /

University of Tokyo, 1996.

[WGM88] André Weinand, Erich Gama e Rudolf Marty. An Object-Oriented Application Framework in C++. *OOPSLA'88- ACM Sigplan Notices.* 23(11), Novembro, 1988.

[WGM89] André Weinand , Erich Gama e Rudolf Marty. Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured programming*, 10(2), 1989.

[Wu97] Z. Wu e S. Schwiderski. Reflective Java: making Java even more flexible. FTP: Arquitecture Projects Management Limited (apm@ansa.co.uk), Cambrige, UK, 1997.

[Wyo88] T. Watanabe e A. Yonezawa. Reflection in an Object-Oriented Concurrent Language. *OOPSLA'88- Conference Proceedings.* Sigplan Notices, 23(11), Novembro, 1988.

[Yok92] Yasuhiko Yokote. *The Apertos Reflective Operating System: The Concept and Its Implementation.* SCSL -TR-92-014 . Sony Compyter Science Laboratory Inc. Tokyo Japão. Junho, 1992.