UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Compreensão Visual de Frameworks através da Introspeção de Exemplos

por
Marcelo Ricardo Campo

Tese submetida à avaliação como requisito parcial
para a obtenção do grau de
Doutor em Ciência da Computação

Prof. Roberto Tom Price
Orientador

Porto Alegre, março de 1997

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Campo, Marcelo Ricardo

Compreensão Visual de Frameworks através da Introspeção de Exemplos / por Marcelo Ricardo Campo. – Porto Alegre: CPGCC da UFRGS, 1997.

258f : il.

Tese (doutorado) - Universidade Federal do Rio Grande do Sul, Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, BR-RS,1997. Orientador: Price, Roberto Tom.

1. Frameworks Orientados a Objetos 2. Reutilização de Software 3. Compreensão de software 4. Reflexão Computacional 5. Visualização de Software. I. Price Roberto Tom. II.Titulo

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Profa.. Wrana Panizzi

Pró-Reitor de Pós-Graduação: Prof. José Carlos Ferraz Hennemann

Diretor do Instituto de Informática: Prof. Roberto Tom Price

Coordenador do CPGCC: Prof. Flávio Wagner

Bibliotecária-Chefe do Instituto de Informática: Zita Prates de Oliveira

Agradecimentos

Escrever um agradecimento para quem deve tanto, a tantos, é uma tarefa tão difícil quanto compreender um framework. Foram muitas as pessoas que me deram seu apoio e colaboraram, direta e indiretamente, para fazer mais fácil o esforço de abandonar o lar para realizar meus estudos de pós-graduação e esta tese, bem como converter esta passagem pelo Brasil na experiência de vida mais valiosa e inesquecível que já tive.

Em primeiro lugar devo agradecer a meu amigo Álvaro Ortigosa por toda a ajuda, compreensão e tolerância durante os anos que moramos aqui juntos e, fundamentalmente, por continuar sendo uma fonte constante de incentivo intelectual e pessoal. Sem sua colaboração desinteressada tudo houvesse sido muito mais difícil.

Devo agradecer à Faculdade de Ciências Exatas da Universidade de Tandil, pelo apoio dado para realizar meus estudos de pós-graduação neste anos. Particularmente, ao Dr. Roberto Gratton e, muito especialmente, à Laura Elissondo, por todo o apoio institucional e pessoal que deram para mim e meu grupo em Tandil, acreditando em nossas capacidades.

Um capítulo aparte merecem "Jony" Orosco e Juan Pablo Solé por todas as discussões, trabalho conjunto e momentos que me fizeram aprender muito o que realmente significa uma interação humana e intelectual com *excelentes pessoas*. Particularmente Juan Pablo, quem faleceu cedo demais, deixou um vazio que nunca preencherei.

Meus orientandos, e amigos, "Piti" Marcos e Mariano Cilia mudaram minha vida quando enviaram, *sem minha permissão*, aquela carta que eu tinha esquecido e que me trouxe a fazer meu doutorado aqui no Brasil. Hoje só posso dizer, obrigado por não ter me respeitado...

Meu "velho" amigo Esteban Pastor deu uma grande força tomando meu lugar em Tandil, durante o primeiro ano, para eu poder afastarme das minhas obrigações docentes.

Meus antigos professores, agora amigos e colegas, Billi Delbue, Hugo Moruzzi, Jane Pryor e Viviana Rubinstein me deram um suporte invalorável durante estes anos, tanto no aspecto pessoal quanto institucional.

Um agradecimento muito especial é para Jorge Boria quem foi minha fonte de inspiração e guia durante minha iniciação na pesquisa e a docência, além de ser a pessoa que mais tem influido e marcado minha vida pessoal e acadêmica. Sem seu suporte (e paciência) tudo teria sido muito mais difícil, ou impossível.

Sou muito grato com os professores do Instituto de Informática e, particularmente, com a Profa. Lúcia Lisboa, por ter inspirado parte de meu trabalho em aquela palestra sobre reflexão computacional, além da sua predisposição permanente para aperfeiçoar meu "portunhol".

O Prof. Alain Pirote da Universidade de Louvaine la Nueve, Bélgica, contribuiu muito com suas críticas e conselhos acerca do meu trabalho e minha carreira. Wolfgang Pree da Universidade Linz, Austria, contribuiu com suas críticas e sugestões. Também, Guillermo Arango contribuiu muito discutindo comingo, há alguns anos, várias idéias que foram de grande valor para meu trabalho.

Aos meus alunos na Universidade de Tandil que com tanto entusiasmo colaboraram para exercitar e melhorar Luthier, agradeço com o mesmo entusiasmo. Aqueles que participaram nos experimentos ajudaram muito com seu trabalho sério para realizar as medições das ferramentas. Meus orientandos colaboraram muito para melhorar e exercitar as implementações desta tese, Alfredo, Camel, Pablito e "Jaureche", e ,especialmente, Patricia e Dorita, por sua dedicação na utilização de Luthier.

Os funcionarios e colegas do Instituto ajudaram muito para me fazer sentir em casa durante estes anos. Particularmente, sou muito grato com Volnei e Juliana por todos os favores, com Rosalvo e sua esposa Rose pela amizade e todos os bons momentos que pasamos juntos, e, finalmente, Júlio por seus esforços para tornar este texto legível.

Concordo com Humberto Eco quando diz que agradecer ao orientador não é de bom gosto (pois é obvio), mas devo agradecer, muito especialmente, à sua família por ter aberto tantas vezes as portas de seu lar para Analia e para mim.

A gratidão com a minha família não pode ser expressada com simples palavras...

A todos, *"muchas gracias"*...

M.R.C.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

CLOS	Common Lisp Object System
OMT	Object Modelling Technique
MIP	Meta-Information Protocol
MO	Meta-Objeto
MOM	Meta-Object Manager
MOP	Meta-Object Protocol
NCM	Nested Context Model

Resumo

Os *frameworks orientado a objetos* oferecem um grande potencial para aumentar a produtividade e a qualidade no desenvolvimento de software. Um framework é uma infra-estrutura ou esqueleto de uma família de aplicações pertencentes a um domínio determinado. Basicamente, aplicações específicas são construídas especializando as classes do framework para fornecer a implementação de alguns métodos, enquanto a maior parte da funcionalidade da aplicação é herdada.

Esta característica permite a reutilização tanto do código quanto o projeto das aplicações do domínio, o qual representa um benefício muito significativo a respeito de outras tecnologias de reutilização. Entretanto, começar a utilizar um framework para construir aplicações específicas é complicado para um usuário que não seja o projetista do framework.. *Compreender* um framework é freqüentemente muito mais difícil que compreender bibliotecas de componentes que podem ser reutilizados independentemente. Neste caso, é suficiente compreender sua interface externa. No caso dos frameworks, para aproveitar ao máximo as possibilidades de reutilização que oferece, é necessário compreender o projeto interno de suas classes, como essas classes colaboram entre si, bem como a forma na qual instâncias dessas classes colaboram em tempo de execução. Compreender estes aspectos é uma tarefa reconhecidamente complexa e demorada, sendo este é um dos fatores mais limitantes da tecnologia para ser de utilidade efetiva na produção de software.

Neste trabalho, apresenta-se uma abordagem reflexiva para a construção de ferramentas de apoio à compreensão de frameworks. Esta abordagem é suportada por Luthier, um framework projetado, e implementado em Smalltalk-80, para a construção de ferramentas de análise dinâmica e visualização de programas orientados a objetos. Luthier introduz três contribuições importantes:

- Utilização de técnicas de reflexão computacional baseadas no conceito de *gerenciadores de meta-objetos*, o qual suporta a implementação de meta-arquiteturas de meta-objetos especializadas para a análise de aplicações.
- Controle interativo do grau de detalhe das visualizações (*zoom semântico*) através do suporte explícito de *escalas de abstração*. As escalas de abstração são controladas por objetos denominados *abstratores*. Um *abstrator* permite encapsular em objetos específicos algoritmos de derivação de abstrações, filtragem e seleção de informação, bem como o controle do nível de detalhe mostrado pelas visualizações. Esta separação de funcionalidade permite implementar complexas funcionalidades de análise de programas, sem a necessidade de modificar as classes que implementam visualizações ou a representação da informação.
- Suporte de uma *estratégia de análise orientada pela visualização da arquitetura*. Esta abordagem divide o processo de compreensão em duas fases iterativa: 1) compreensão dos principais aspectos estruturais do framework a partir de da recuperação e visualização da arquitetura,; 2) análise detalhada do comportamento de instâncias envolvidas em pontos específicos da arquitetura, os quais são selecionados pelo usuário a partir da visualização arquitetônica.

Luthier fornece suporte flexível para construção de ferramentas de visualização *dinamicamente adaptáveis* para diferentes funcionalidades de análise, através de quatro sub-frameworks: *LuthierMOPs*, o qual fornece o suporte adaptável de meta-objetos para captura de informação das aplicações analisadas; *LuthierBooks* que fornece suporte genérico de gerenciamento de hiperdocumentos para a representação da informação capturada e gerenciamento de livros persistentes de projeto; *LuthierAbstractors*, que provê suporte genérico para a derivação de abstrações da informação coletada e escalas de abstração dinamicamente variáveis; e *LuthierViews*, extensão do framework MVC para a construção de visualizações da informação coletada, com capacidades de

manipulação direta e *zooming* utilizando visualizações alternativas, as quais podem ser dinamicamente selecionadas pelo usuário.

Com o suporte fornecido por Luthier, uma ferramenta, especialmente projetada para apoiar a compreensão de frameworks a partir da análise de exemplos, foi desenvolvida. Esta ferramenta fornece um conjunto de visualizações estruturais, com capacidade de animação de fluxo de controle do framework, bem como visualizações alternativas de subsistemas e padrões de projeto. Estas abstrações são reconhecidas através da análise da informação coletada dos exemplos analisados. Através das visualizações providas, o usuário pode explorar um dado framework através de mecanismos de navegação entre diferentes representações visuais, bem como filtragem e consulta acerca de informação relevante a ser visualizada. Este mecanismos são integrados com mecanismos de *zoom semântico* que habilitam a visualização da informação em diferentes níveis de abstração. Através da representação de hiperdocumento, a ferramenta suporta a construção manual, bem como a geração automática em alguns casos, de livros persistentes de documentação, com capacidade de edição, de importação de diagramas produzidos pelas visualizações, e de navegação sobre diferentes livros e o código fonte do framework. Esta caraterística habilita a geração de documentação durante o processo de compreensão, facilidade não disponivel, habitualmente, nas ferramentas de compreensão desenvolvidas até hoje.

A capacidade de Luthier para a construção de ferramentas foi testada com a construção de outras ferramentas, como por exemplo, depuradores visuais e de coleta de métricas. A viabilidade e eficácia da abordagem foi testada através de experimentos, os quais mostraram que grupos utilizando a ferramenta de apoio produziram aplicações com maior nível de reutilização do framework que grupos de usuários não utilizando a ferramenta.

Palavras-chave: Reutilização de Software, Frameworks Orientados a Objetos, Visualização de Software, Reflexão Computacional.

TITLE: "VISUAL UNDERSTANDING OF FRAMEWORKS THROUGH INSTROSPECTION OF EXAMPLES".

Abstract

Object-oriented frameworks are a powerful reuse technique for building applications in a given domain. A framework works as a template or skeleton for building applications, being composed of a set of classes abstracting the general characteristics of an application domain. Building a specific application requires the specialization of some classes that provide the implementation of methods that will complete the necessary behaviour, while the global control structure is given by the framework.

Frameworks offer a great potential to increase the productivity and quality in software development. However, starting to use a framework to build a specific application is complicated for any user other than a framework designer. Understanding a framework is frequently much harder than understanding libraries of components that can be reused independently. To adequately reuse isolated classes of a class library, it is sufficient to understand their external interface. In case of a framework, in contrast, to take full advantage of the services provided, it is necessary to understand the way its classes collaborate, as well as the internal design of some of them. These classes code the complex behaviour of a network of instances dynamically created. Therefore, it is often needed not only to understand how the classes are organized in static inheritance hierarchies, but also how instances collaborate at runtime.

This work presents a reflective approach for the construction of tools for framework comprehension. This approach is supported by Luthier, a framework designed, and implemented in Smalltalk-80, which provides a flexible support for building tools for the dynamic analysis and visualization of object-oriented programs. Luthier introduces three important contributions:

- The use of computational reflection techniques, based on the concept of *meta-object managers*, which support the implementation of specialized meta-object-based meta-architectures for the dynamic analysis of applications.
- Support for building visualizations with different levels of abstraction under interactive control of the user (*semantic zoom*) through the explicit support of *abstraction scales*. Abstraction scales are controlled by objects called *abstractors*, which allow the encapsulation, in specific objects, of algorithms for abstraction derivation, filtering and information selection, as well as the interactive control of the detail level to be shown by visualizations. This separation of concerns enables the implementation of complex program analysis functionalities without the need of modifying classes implementing visualizations or information representation.
- Support for an architecture-driven analysis strategy. This approach divides the understanding process in two iterative phases: 1) comprehension of the global structural and behavioural aspects of the framework from the recovery and visualization of its architecture.; 2) detailed analysis of specific instances involved in particular points of the architecture, selected by the user form the architectural view.

Luthier provides a flexible support for the construction of visualization tools dynamically adaptable to different analysis functionalities through four sub-frameworks: *LuthierMOPs*, which provides an adaptable support of meta-objects for information gathering from the analyzed applications; *LuthierBooks*, which provides generic support for hyperdocument management to represent captured information and management of persistent design books; *LuthierAbstractors*, which provides generic support for the derivation of abstractions and dynamically variable abstraction-scales; and *LuthierViews*. extension of the MVC framework for the construction visualizations of the captured information, with capabilities of direct manipulation and *zooming* using alternative visualizations, dynamically selected by the user.

With the support provided by Luthier a tool, specially designed to support framework comprehension from analysis of examples, was developed. This tool provides a set of structural visualizations with control-flow animation capabilities, as well as alternative visualizations of subsystems and design patterns, recognized through the analysis of information gathered from examples. Through these visualizations the user can explore a given framework by using mechanisms for navigating among different visual representations, as well as information filtering and queries about relevant information to be visualized. These functionalities are fully integrated with semantic zoom mechanisms that enable information visualization at different levels of abstraction. With the hyperdocument support, the tool allows for manual construction, as well as automatic generation in some cases, of persistent documentation books. These books offers editing capabilities, importation of diagrams from visualizations, as well as navigation through different books and through the source code of the analyzed framework.. These characteristics allows the support of additional documentation generation during the comprehension process, facility which is not normally available in current understanding tools.

The capabilities for tool construction supported by Luthier were tested through the development of different tools, such as, visual debuggers and metrics collectors. The viability of the approach was tested through experiments. These experiments suggest that users using the understanding tool produce applications with a greater re-use level than groups of users not using it..

Keywords: Software Reuse, Object-Oriented Frameworks, Software Visualization, Computational Reflection.

1 Introdução

Na medida que as aplicações de software tornam-se mais complexas, tecnologias e ferramentas de apoio que habilitem a reutilização do projeto dessas aplicações tornam-se, progressivamente, mais necessárias. O projeto de um sistema é uma das tarefas mais importantes dentro do ciclo de desenvolvimento de software. O projeto de um sistema define, essencialmente, sua divisão em partes, as interfaces entre elas e como essas partes cooperam para implementar a funcionalidade requerida. Um projeto adequado facilita mudanças e a evolução do sistema, minimizando o impacto de mudanças locais sobre outras partes do sistema, quando deve ser adaptado para novos requisitos. O projeto de um sistema, é, também, uma das tarefas mais complexas dentro do ciclo de desenvolvimento. Winograd carateriza o projeto de software como uma atividade interdisciplinar. Em primeiro lugar, projeto envolve os aspectos de organização de um sistema desde o ponto de vista da programação de componentes que resolvem um dado problema. Projeto, por outro lado, também envolve aspectos cognitivos das interfaces com o usuário a ser providas, caraterísticas específicas de plataformas hardware que suportarão a implementação, bem como conhecimento específico acerca do domínio da aplicação [WIN 96]. Considerando-se esta diversidade de fatores, é razoável afirmar que desenvolver um projeto que forneça um equilíbrio adequado entre todos estes aspectos, não é uma tarefa que possa ser facilmente realizada por projetistas inexperientes [DEU 89][JOH 93[GAM 94]. Portanto, a *reutilização de projetos existentes* torna-se muito importante.

Reutilizar o projeto de uma aplicação existente permite orientar o desenvolvimento de uma aplicação, induzindo decisões adequadas *a-priori*, que diminuem a necessidade de desfazer decisões erradas que poderiam ser tomadas em etapas iniciais do projeto [COC 91]. Se essa aplicação foi projetada por especialistas no domínio, reutilizar seu projeto implica, portanto, na reutilização da experiência e conhecimento que esses especialistas colocaram na resolução dos problemas desse domínio. Deutsch aponta que replicar este processo de projeto, é, definitivamente, mais difícil que replicar o processo de implementação de uma aplicação [DEU 89]. Assim, se a informação de projeto está disponível e convenientemente codificada, o processo de resolução de problemas fica reduzido à resolução de aspectos específicos da aplicação sendo desenvolvida [BIG 87].

Com o advento da orientação a objetos, nos começos da década passada, surgira um novo paradigma de desenvolvimento no qual a reutilização é parte integral do paradigma. Os mecanismos de herança, polimorfismo e acoplamento dinâmico, introduziram a tecnologia habilitante para a construção de software por especialização, em base à extensão de funcionalidade de componentes existentes. Através destes mecanismos é possível definir componentes abstratos, ou semi-acabados, cujo comportamento específico pode ser completado por subclasses. Deste modo, na criação de componentes específicos, é possível reutilizar tanto o projeto definido pelos componentes abstratos quanto o código que implementa esse comportamento.

A capacidade de definição de comportamentos abstratos, pode ser generalizada para codificar o comportamento genérico de uma família de aplicações. Cada aplicação da família, pode ser construída através da implementação de porções de comportamento que são específicos a ela. Assim, é possível reutilizar além de classes isoladas, classes projetadas em conjunto para resolver a funcionalidade genérica de um domínio de aplicação. Isto significa, definitivamente, reutilizar o projeto de aplicações.

Um conjunto de classes que fornecem uma solução genérica para um dado domínio de aplicação é denominado de ***framework orientado a objetos***[1] [DEU 89][JOH 88].

Um framework é constituído por um conjunto de classes que definem um projeto abstrato para soluções de problemas de uma família de aplicações dentro de um domínio [JOH 88]. Um

[1] O termo português equivalente para framework é arcabouço, mas neste texto utiliza-se o termo inglês para manter consistência com a bibliografia existente.

framework implementa, em termos de classes, o comportamento genérico de um domínio de aplicação, deixando a implementação dos aspectos específicos de cada aplicação para serem completados por subclasses. Deste modo, um framework funciona como um *molde* para a construção de aplicações, ou subsistemas, dentro de um domínio de aplicação.

Johnson caracteriza os *frameworks* como induzindo um processo desenvolvimento de aplicações dominado pela *inversão de controle* [JOH 88]. Na reutilização de componentes de biblioteca (funções, procedimentos ou classes) um programador deve desenvolver a estrutura de controle que invoca esses componentes; utilizando um framework, ao contrário, o programador deve desenvolver código que será invocado pelo framework. A estrutura de controle da aplicação é codificada pelas classes do framework, as quais invocam métodos que devem ser implementados para cada aplicação particular. Isto é, aplicações específicas são construídas especializando as classes do *framework* para fornecer a implementação de alguns métodos, enquanto a maior parte da funcionalidade da aplicação é herdada.

Esta característica representa um benefício significativo, pois, uma vez que o framework tenha sido compreendido, os desenvolvedores devem concentrar-se na solução dos aspectos específicos do problema sendo resolvido. Esta característica, também explica porque a reutilização baseada em frameworks é considerada como reutilização de projeto. As aplicações desenvolvidas utilizando um framework possuem todas a mesma estrutura e seus componentes respondem todos ao mesmo protocolo, diferenciando-se umas das outras pelo comportamento de métodos específicos. Deste modo, se um framework é projetado por especialistas em um domínio, usuários que utilizam o framework reutilizam, implicitamente, a experiência destes projetistas.

A utilização de frameworks no desenvolvimento de software pode habilitar um maior nível de reutilização, tanto de código como de projeto, do que de outras tecnologias de reutilização. Tecnologias como 4GLs, geradores de código ou mesmo bibliotecas de classes, úteis sem dúvida alguma, entretanto, são geralmente limitadas a domínios restritos, e não favorecem a construção de infra-estruturas reutilizáveis como as possíveis através dos frameworks. Esta infra-estrutura já fornece grande parte da funcionalidade necessária para construir uma aplicação, reduzindo, em conseqüência, os esforços de programação, manutenção, teste e depuração. Isto, naturalmente, implica em um aumento da *produtividade* no desenvolvimento, mas, ao fornecer uma estrutura de projeto que é reutilizada implicitamente quando constrói-se uma aplicação, a *qualidade* da aplicação desenvolvida também pode ser consideravelmente melhorada.

Através de interfaces adequadamente projetadas, os frameworks oferecem um alto grau de flexibilidade para estender a funcionalidade de aplicações, acrescentando comportamento, sem sacrificar a compatibilidade e interoperabilidade dos componentes. Isto impacta também, na manutenibilidade das aplicações, pois devido à herança, erros corrigidos ou novas funcionalidades agregadas, propagam-se automaticamente às classes derivadas, minimizando o impacto dessas mudanças no resto das classes.

1.1 O Problema

Tomando em consideração as vantagens descritas acima, os *frameworks* oferecem um grande potencial para aumentar a produtividade e a qualidade no desenvolvimento de software. Entretanto, começar a utilizar um *framework* para construir aplicações específicas é complicado para um usuário que não seja o projetista do framework.

Compreender um framework é, freqüentemente, muito mais difícil que compreender bibliotecas de componentes que podem ser reutilizados independentemente. Neste caso é suficiente compreender suas interfaces externas. No caso dos frameworks, para aproveitar ao máximo as possibilidades de reutilização que oferece, é necessário compreender o projeto interno de suas classes, como essas classes colaboram entre si, bem como a forma na qual instâncias dessas classes colaboram em tempo de execução [LAJ 94][LAN 95].

Alcançar a compreensão detalhada destes aspectos é, geralmente, uma tarefa custosa e demorada. Este representa o fator mais limitante da tecnologia. Booch define claramente esta situação:

"um framework pode ser realmente útil para desenvolver aplicações só se o custo de compreender e utilizar suas abstrações é muito menor que o custo percebido pelo programador para desenvolver a aplicação desde zero " [BOO 94].

O problema da compreensão de frameworks pode ser atribuído, em primeiro lugar, à complexidade inerente de compreensão de programas orientados a objetos, amplamente relatada na literatura [HEL 90][WIL 92][JOH 92][DEP 93][DEP 94][LAJ 94][LAN 95]. Mas, também as limitações para representar projetos abstratos das técnicas atuais de documentação de projeto orientado a objetos, representam um outro fator altamente limitante [CAS 92][BUH 93][JOH 92].

Por esta razão, várias técnicas especiais de documentação, como *contracts* [HEL 90], *patterns* [JOH 92] e *meta-patterns* [PRE 94], tem sido propostas para ajudar a um usuário a compreender como um dado framework é organizado. Algumas delas também descrevem como o framework pode ser utilizado para construir aplicações. Sem dúvida alguma, estas técnicas são muito úteis para documentar frameworks estáveis e maduros. No entanto, nem todos os frameworks são documentados utilizando estas técnicas, e ainda que o sejam, examinar aplicações ou exemplos desenvolvidos utilizando um *framework* é, geralmente, um bom ponto de início para compreender como suas classes são instanciadas e relacionadas umas com outras.

Alcançar uma compreensão detalhada do framework através da inspeção do código das suas classes é uma tarefa custosa, quando não quase impossível, se o *framework* é muito complexo e não existe documentação apropriada. Assim, métodos de documentação eletrônica baseada em exemplos, tal como os *cookbooks* de Smalltalk, fornecem um veículo de grande valor para assistir no processo de utilização de um framework para construir aplicações. Esta abordagem, porém, sofre da limitação imposta pela impossibilidade de prever todos os potenciais usos de um framework, limitando-se, no caso geral, a exemplos prototípicos da forma de utilizar o framework nas situações mais comuns.

Neste contexto, ainda que haja documentação disponível, ferramentas capazes de analisar o comportamento de aplicações e exemplos providos por um framework, e visualizar como suas classes se organizam e relacionam através da troca de mensagens, tornam-se um complemento muito importante para facilitar a compreensão desse framework e, portanto, produzir aplicações mais rapidamente maximizando a reutilização de funcionalidade provida pelo framework.

Nos últimos anos várias ferramentas que objetivam auxiliar no processo de compreensão de software orientado a objetos tem sido relatadas na literatura. Estas abordagens, centram-se, essencialmente, na provisão de visões microscópicas do comportamento de um programa com objetivos de depuração [BRU 93][STA 94][VIO 94], ou na provisão de mecanismos alternativos de visualização da informação do programa [DEP 93][DEP 94]. Mas, pouco trabalho foi desenvolvido para suportar especificamente a compreensão de frameworks.

As ferramentas, desenvolvidas até hoje caraterizam-se pela adaptação dos mecanismos de captura de informação desenvolvidos para os sistemas de visualização de algoritmos, introduzidos pelo sistema BALSA [BRO 84], os quais implicam na necessidade de modificar o código das aplicações. Além disso, raramente fornecem capacidades para construção tanto de novas visualizações quanto de abstrações da informação coletada. Isto limita sua adaptabilidade para diferentes níveis de abstração e tipos de usuários, reduzindo, em conseqüência, sua utilidade.

1.2 A Tese

No contexto descrito acima, a hipótese essencial desta tese é que ferramentas visuais para apoio na análise de aplicações construídas utilizando um dado framework tornam-se um complemento essencial para explorar as vantagens que os frameworks oferecem para a produção de software.

Através da provisão de mecanismos de visualização adequados, estas ferramentas podem facilitar em muito a compreensão detalhada de um dado framework, permitindo, em

conseqüência, produzir aplicações mais rapidamente, maximizando a reutilização de funcionalidade nele existente.

Sob esta hipótese, surgem quatro aspectos básicos a serem considerados para estas ferramentas atenderem adequadamente o processo de compreensão de *frameworks*:

- Qual o mecanismo adequado, adaptável a linguagens industriais como C++ e Smalltalk, para suportar a análise estática e dinâmica de ditas aplicações e recuperar a informação relevante a ser visualizada acerca do *framework*, minimizando a necessidade de modificar o código dessas aplicações ?
- Qual a forma de representar esta informação que habilite tanto a derivação de abstrações quanto a geração de documentação durante o processo de compreensão ?
- Quais os mecanismos que permitam adaptar as visualizações para facilitar a exploração da informação desde diferentes perspectivas e níveis de detalhe, minimizando a necessidade de programar visualizações especializadas ?
- Qual o nível de abstração adequado das visualizações que devem ser providas para facilitar a compreensão de *frameworks*, tanto por parte de usuários especialistas quanto não especialistas ?

Com o objetivo de fornecer um suporte extensível para resolver estes aspectos foi projetado, e implementado em Smalltalk-80, o framework Luthier [CAM 96a] para a construção de ferramentas de análise dinâmica e visualização de frameworks orientados a objetos.

Luthier fornece suporte flexível para construção de ferramentas de visualização, dinamicamente adaptáveis para diferentes funcionalidades de análise, utilizando um gerenciador de hiperdocumentos para organizar a informação coletada. Esta representação suporta tanto a construção de diferentes visualizações dos exemplos analisados, bem como a navegação entre diferentes representações e a documentação do processo de compreensão desses exemplos, através do suporte para a edição de livros de documentação.

Do ponto de vista do projeto de ferramentas, Luthier introduz duas contribuições principais:

- utilização de técnicas de reflexão computacional baseadas no conceito de *gerenciadores de meta-objetos* [CAM 96b], o qual suporta a implementação de meta-arquiteturas de meta-objetos especializadas para a análise dinâmica de aplicações. Através desta meta-arquitetura, a informação estática e dinâmica dos relacionamentos entre classes e instâncias pode ser coletada por meta-objetos que interceptam o normal fluxo de mensagens entre os objetos da aplicação.
- controle interativo do grau de detalhe das visualizações (*zoom semântico*) através do suporte explícito de *escalas de abstração*, controladas por objetos denominados *abstratores*. Um *abstrator* permite encapsular em objetos específicos algoritmos de derivação de abstrações, filtragem e seleção de informação, bem como o controle do nível de detalhe mostrado pelas visualizações, sem necessidade de modificar ou implementar funcionalidade específica nas visualizações nem na representação da informação.

Através do suporte de meta-objetos é possível programar meta-objetos específicos para extrair informação estática e dinâmica de aplicações ou exemplos desenvolvidos com um dado framework, e construir uma representação abstrata da estrutura desse framework utilizando um gerenciador de hiperdocumentos. Com base nesta informação, através da implementação de abstratores, diferentes abstrações da informação podem ser deduzidas, como por exemplo, subsistemas e padrões de projeto [GAM 95], bem como visualizações abstratas da estrutura do *framework* analisado podem ser construídas através do suporte de visualização.

A utilização de técnicas reflexivas, combinada com visualizações estruturais, fornece o suporte adequado para uma estratégia iterativa de análise, também introduzida por esta tese, denominada *estratégia de compreensão orientada pela arquitetura*. Esta estratégia divide o processo de compreensão em duas fases iterativas:

- Em uma primeira fase visualizações estruturais guiam o usuário na exploração do *framework* através de mecanismos de navegação sobre múltiplas visões, consulta e filtragem de informação, no nível da organização de classes, métodos e estrutura de controle abstrata e concreta. Através destes mecanismos o usuário pode alcançar uma compreensão detalhada do funcionamento das principais classes do framework, quais as classes que devem ser especializadas e quais métodos implementados.
- Uma vez que os aspectos globais do comportamento e estrutura do framework foram compreendidos, o usuário pode avançar na análise do comportamento das instâncias envolvidas em porções específicas do framework que requerem uma análise a nível de instâncias. Selecionando interativamente aquelas partes que devem ser analisadas através das visualizações estruturais, diferentes meta-objetos são dinamicamente associados com os objetos da aplicação analisada para coletar informação acerca de instâncias específicas ou suspender a execução em métodos selecionados.

Luthier é projetado para fornecer um alto grau de flexibilidade na construção de mecanismos de captura, representação, abstração e visualização da informação, sendo por esta razão, dividido em quatro sub-frameworks:

- *LuthierMOPs*: Suporte de meta-objetos baseado no conceito de *gerenciadores meta-objetos*, para captura de informação das aplicações analisadas.
- *LuthierBooks*: Extensão do framework de suporte de gerenciamento de hiperdocumentos, PROMETO [ORT 95], para a representação da informação capturada e gerenciamento de livros persistentes de projeto.
- *LuthierAbstractors*: Suporte genérico para a derivação de abstrações da informação coletada e escalas de abstração dinamicamente variáveis para definir os diferentes graus de detalhe de cada visualização.
- *LuthierViews*: Extensão do framework MVC [PAR 94] para a construção de visualizações da informação coletada com capacidades de manipulação direta e *zooming* de informação utilizando visualizações alternativas, dinamicamente selecionáveis pelo usuário.

Uma ferramenta típica construída com Luthier estará composta por um conjunto de meta-objetos que monitoram a execução de uma aplicação, gerando uma representação da informação requerida utilizando um gerenciador de hiperdocumentos. O sistema de visualização requererá ao nível inferior a informação a ser visualizada, a qual será provida por abstratores encarregados de selecionar ou construir abstrações da informação contida na representação de hiperdocumento. A FIGURA 1.1 apresenta um esquema do funcionamento e mecanismos de suporte providos por um protótipo para apoio na análise de frameworks, desenvolvido com Luthier.

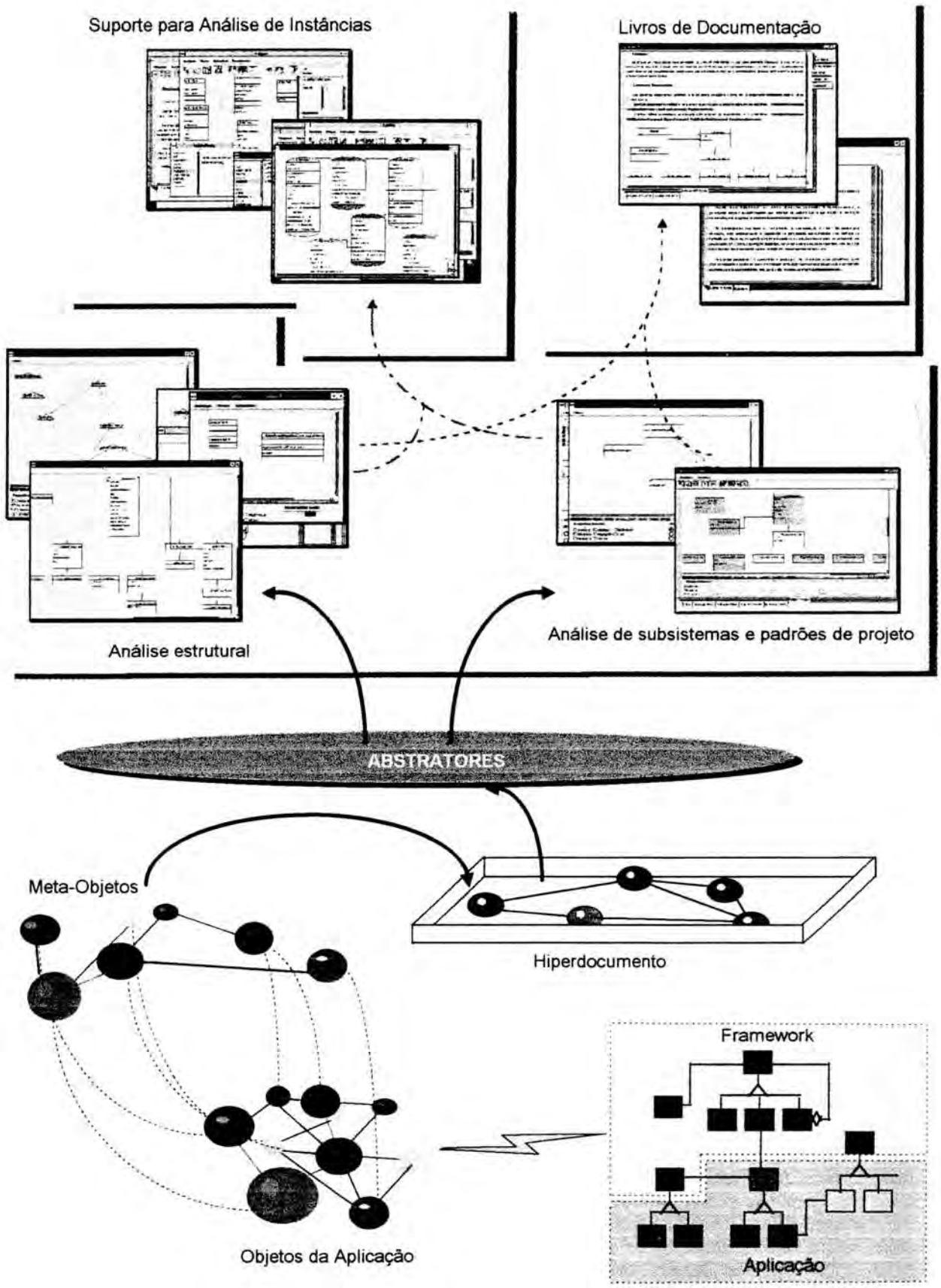


FIGURA 1.1- Esquema genérico de ferramentas construídas com Luthier

1.3 Organização do Texto da Tese

Este relatório está constituído por 9 capítulos e três anexos, com a organização descrita a seguir.

No capítulo 2 apresenta-se uma descrição geral dos aspectos estruturais que caraterizam os frameworks orientados a objetos e analisa-se o relacionamento entre os conceitos de modelo de domínio e arquiteturas de software, fornecendo uma definição de framework em termos destes conceitos.

No capítulo 3 analisa-se o problema da compreensão de frameworks desde a perspectiva da compreensão de software em geral e do caso particular do software orientado a objetos. Neste capítulo exemplifica-se esta problemática através de um exemplo real de uso do framework MVC e analisam-se as vantagens e limitações das técnicas de documentação de software orientado a objetos para a documentação de frameworks, bem como das técnicas específicas desenvolvidas para documentar frameworks, existentes na literatura.

O capítulo 4 apresenta uma descrição das técnicas de apoio à compreensão de software desde a ótica das contribuições da área de engenharia reversa e sistemas de visualização de software. Neste contexto, identificam-se quatro áreas chave que definem um modelo de domínio para as ferramentas de apoio à compreensão, analisam-se as principais abordagens dentro de cada área adotadas pelas ferramentas de apoio à compreensão de software orientado a objetos existentes na literatura, e se descrevem sinteticamente as soluções adotadas neste trabalho para cada uma dessas áreas.

No capítulo 5, apresenta-se uma descrição geral dos principais conceitos e modelos de reflexão computacional na orientação a objetos. Neste capítulo discutem-se a diferentes alternativas de implementação para dotar uma linguagem não reflexiva com suporte de meta-objetos, e em base a esta discussão apresenta-se o conceito de *gerenciadores de meta-objetos* introduzido neste trabalho.

No capítulo 6, realiza-se a descrição geral e a especificação formal de Luthier. Esta descrição é realizada mostrando as caraterísticas mais relevantes dos quatro sub-frameworks em termos informais, com exemplos de código e diagramas de estruturas e interação, e a especificação formal dos contratos que definem o projeto detalhado do framework utilizando a notação de contratos [HEL 90].

No capítulo 7, descrevem-se as características principais de um protótipo de ferramenta de visualização desenvolvida com Luthier para apoio na compreensão de frameworks. Esta descrição é realizada utilizando o próprio Luthier como exemplo para mostrar as diferentes funcionalidades providas para a visualização, exploração e documentação de frameworks

O capítulo 8 apresenta resultados de experimentos realizados com este protótipo para verificar sua utilidade como ferramenta de apoio à compreensão e descreve, brevemente, uma ferramenta de coleta de métricas desenvolvida com Luthier.

No capítulo 9, apresentam-se as conclusões da tese e discutem-se outras possíveis extensões a Luthier e áreas de futura pesquisa.

O anexo 1 descreve sinteticamente a notação utilizada para especificação formal de Luthier. O anexo 2 apresenta uma descrição sintética do catálogo de padrões de projeto, para aqueles leitores que não tenham um conhecimento prévio destes padrões. Finalmente, o anexo 3, apresenta uma descrição resumida das características principais da linguagem Smalltalk.

2 Frameworks Orientados a Objetos

O que é um framework? Qual a diferença entre um framework e uma hierarquia de classes? Quando uma hierarquia de classes pode ser considerada um framework? Estas questões surgem, freqüentemente, quando é necessário explicar o conceito de framework e contrastá-lo com outras tecnologias de reutilização. Estas questões também são importantes para definir com exatidão os requisitos que uma ferramenta de apoio deve satisfazer para servir eficientemente ao processo de compreensão de frameworks. Neste capítulo analisam-se estas questões através da caracterização dos frameworks, do ponto de vista da sua estrutura interna como também dos mecanismos que fornecem para a construção de aplicações, definindo o conceito de framework em termos de sua relação com os conceitos de modelo de domínio, arquitetura de software e capacidade de geração de aplicações.

2.1 Reutilização na Orientação a Objetos

Para definir reutilização de software é necessário diferenciar, em primeiro lugar, o que significa reutilizar um dado artefato em contraste com utilizar esse artefato. No caso geral, quando se fala de utilização, está implícito o fato que o artefato é utilizado sem realizar nenhuma mudança tanto na sua estrutura quanto na forma e contexto no qual é utilizado. Quando um dado artefato é utilizado após realizar-se alguma mudança ou bem é utilizado para uma tarefa ou função diferente para a qual foi originalmente projetado então esse artefato está sendo efetivamente reutilizado. Assim, entende-se por reutilização de software quando um artefato de software é utilizado novamente variando sua forma original ou o contexto (aplicação) no qual está sendo reutilizado [DEU 89]. Esta definição coloca em evidência a caraterística intrínseca da reutilização de software que é a necessidade de adaptação dos artefatos para adequá-los aos requisitos da nova aplicação sendo desenvolvida.

Aprofundando na natureza da reutilização, podem distinguir-se duas formas básicas de reutilização de software: reutilização de porções de código e reutilização de unidades funcionais completas, as quais podem tomar a forma de procedimentos, pacotes ou módulos. No primeiro caso, segmentos de código desenvolvidos para uma aplicação são incorporados em um novo artefato de software depois de serem adaptados. No segundo caso, o código é reutilizado através de uma interface previamente estabelecida. Esta interface é determinada pela especificação dos serviços que o artefato fornece aos seus clientes, em termos dos tipos dos seus argumentos de entrada e o tipo de resultado que retorna. Os clientes do componente referenciam esta interface no seu código de acordo com a especificação de argumentos. Com esta estratégia tanto a implementação do cliente como a implementação da interface pode variar sem afetarem-se reciprocamente.

Ambas formas de reutilização podem ser suportadas por diferentes tecnologias de implementação. Um forma bastante comum é a clonagem [GOL 95]. Diferentes reutilizadores realizam cópias do artefato a ser reutilizado e o adaptam para os requisitos da aplicação sendo desenvolvida. A clonagem pode oferecer vantagens em termos de produtividade, mas não do ponto de vista da manutenibilidade dos artefatos produzidos. A replicação de código pode gerar diferentes versões que variam de uma aplicação para outra e que não necessariamente são refletidas no artefato original. Deste modo, o benefício da reutilização fica reduzido só a aspectos de desenvolvimento, mas não de qualidade do software desenvolvido.

As linguagens orientadas a objetos fornecem capacidades para a reutilização seletiva de interface e implementação, através da combinação dos mecanismos de herança, polimorfismo e acoplamento dinâmico.

Uma classe define a estrutura interna e a implementação dos serviços de um conjunto de objetos, os quais são as suas instâncias. Através do mecanismo de herança é possível adaptar seletivamente o comportamento de alguns serviços redefinindo sua implementação em subclasses. Deste modo é possível reutilizar a implementação de um componente, mantendo a interface invariável. É importante distinguir aqui entre reutilização de interface e reutilização de implementação. Através do

mecanismo de herança, uma subclasse herda a implementação tanto da estrutura interna como dos serviços definidos nas superclasses. Os serviços definidos pela interface definem, por sua vez, o *tipo* do objeto. A herança de interfaces determina que uma subclasse seja um *subtipo* do tipo definido pela superclasse. Este relacionamento permite que instâncias de diferentes subclasses possam ser trocadas para mudar dinamicamente o comportamento de uma aplicação. Os clientes de uma classe só devem fazer referência aos serviços definidos pelo tipo, evitando desta forma a dependência de implementações específicas.

Dependendo da linguagem de programação os conceitos de tipo e classe variam. Algumas linguagens, como por exemplo Java, fornecem os mecanismos para diferenciar explicitamente entre interface e classe, o qual permite expressar em forma independente o tipo do objeto e a sua implementação. Em linguagens tipadas, como por exemplo C++ ou Eiffel, a classe define tanto o tipo quanto a implementação do objeto. Em linguagens não tipadas, como Smalltalk, a classe define só a implementação de um objeto. Neste caso um objeto pode, conceitualmente, pertencer a vários tipos diferentes, também como um objeto pode ser substituído por um outro qualquer sempre que a classe deste último implemente a mesma interface.

2.1.1 Reutilização por composição e por herança de classes

Os mecanismos de reutilização de interface e implementação discutidos nas seções precedentes habilitam duas formas alternativas de reutilização de funcionalidade em sistemas orientados a objetos: herança de classes e composição de instâncias

O mecanismo de herança habilita a construção de novos componentes através de subclasses que herdam a implementação da estrutura e serviços definido nas superclasses. Este tipo de reutilização é também denominado ***caixa-branca***, pois as subclasses tem acesso à implementação herdada.

A reutilização por composição de instâncias, por sua vez, coloca a ênfase na distribuição de funcionalidade básicas entre diferentes objetos, os quais são combinados ou compostos para obter diferentes comportamentos. Este tipo de reutilização é denominado ***caixa-preta***, pois os diferentes objetos podem ser reutilizado através da sua interface, ou seja, não é necessário que os clientes externos conheçam a estrutura e a implementação dos serviços de cada componente.

Cada forma de reutilização possui vantagens e desvantagens. A reutilização baseada em herança de implementação permite construir com pouco esforço novos componentes através de subclasses. Estes componentes compartilham a maior parte da funcionalidade, o qual permite propagar automaticamente às subclasses as mudanças realizadas nas superclasses. Esta vantagem, entretanto, também representa uma desvantagem. No caso geral, as subclasses herdam a estrutura interna das superclasses, o que implica na necessidade de modificar manualmente as subclasses quando uma mudança na estrutura da superclasse é realizada. Uma outra restrição que apresentam estas implementações é a adaptação de comportamento em tempo de execução. Os métodos a serem aplicados são determinados estaticamente, inibindo a possibilidade de utilizar implementações alternativas de um mesmo serviço, como por exemplo, algoritmos de ordenação.

A reutilização baseada em composição, por sua vez, oferece a vantagem de suportar a adaptação dinâmica de aplicações. Sob esta estratégia, as classes implementam funções muito específicas, as quais são utilizadas através de uma interface bem definida. Comportamentos complexos são criados através da composição de diferentes objetos, sendo que cada um deles mantém referências a um ou vários outros objetos. Através destas referências um objeto pode delegar a responsabilidade de executar uma dado serviço a um outro objeto. O objeto, o qual efetivamente realizará a função, depende da configuração de instâncias criada. Assim, por exemplo, diferentes objetos podem implementar diferentes algoritmos de ordenamento, os quais podem ser passados como parâmetros aos objetos que serão os seus clientes.

A reutilização através da composição de objetos é mais desejável que a reutilização baseada em herança, mas, no caso geral, não é possível definir um conjunto adequado de componentes que permitam implementar qualquer funcionalidade dentro de um domínio. Habitualmente, é necessário

desenvolver novos componentes que forneçam serviços não disponíveis. Para isto, a herança serve como o complemento que permite a extensão de funcionalidade para a construção de novos componentes, aproveitando a funcionalidade provida pelos componentes existentes. Deste modo, composição e herança são técnicas complementares que, quando utilizadas apropriadamente, contribuem para o desenvolvimento de sistemas altamente flexíveis e reutilizáveis.

2.2 Classes Abstratas

Independentemente das capacidades fornecidas por uma linguagem, o conceito de herança de interface pode ser implementado através de *classes abstratas*.

Uma classe abstrata é uma classe na qual pelo menos uma operação ou método não está implementado. Por esta razão, uma classe abstrata nunca terá instâncias e será utilizada só como uma superclasse. As classes que não são abstratas são denominadas *concretas*.

Uma classe abstrata é projetada para ser utilizada como um *molde* para especificar subclasses, enquanto uma classe concreta é projetada para especificar objetos. Uma classe abstrata define o conjunto de serviços ou *protocolo* ao qual responderão todos os objetos pertencentes as suas subclasses, através de *métodos abstratos*.

Um método abstrato define a interface de uma operação, mas não fornece uma implementação. Devem ser redefinidos em subclasses para implementar variantes especificas do comportamento esperado do método. As linguagens de programação fornecem diferentes sintaxes para definir estes métodos. Por exemplo, *Eiffel* fornece a qualificação ***deferred*** para diferenciá-los. C++ permite a definição de um método como ***pure virtual*** para indicar que é um método não implementado pela classe. Smalltalk, por sua vez, não fornece nenhuma notação especial, sendo estes definidos pela mensagem ***self subclassResponsibility***, a qual indica que a implementação do método tem que ser fornecida por uma subclasse. Se alguma subclasse não redefine algum destes métodos e um cliente o chama, produzir-se-á um erro de execução informando que esse método não está implementado.

Quando todos os métodos definidos por uma classe abstrata são abstratos, essa classe define efetivamente um *tipo* de objetos. A classe não determina a representação da estrutura interna dos objetos, mas só o comportamento definido pelos serviços representados pelos métodos abstratos. Assim, diferentes subclasses podem definir diferentes implementações tanto para a representação da abstração quanto para a implementação dos serviços. O polimorfismo permite que os clientes definam variáveis que referenciam esse tipo de objetos como do tipo definido pela classe abstrata, evitando a dependência com subclasses concretas especificas. Isto permite aumentar a reutilizabilidade dos clientes, como também a flexibilidade da solução, pois variáveis definidas de um tipo A poderão referenciar objetos de qualquer subtipo de A.

Um exemplo clássico de classe abstrata é a classe *Magnitude* provida pela biblioteca de Smalltalk-80. Esta classe define o protocolo comum que carateriza os objetos que podem ser comparados através das operações básicas >,<,=, etc.

```
Object subclass: #Magnitude
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Magnitude-General'
Magnitude comment:
'The abstract class Magnitude provides common protocol for objects that have the ability to be
    compared along a linear dimension, such as dates or times. Subclasses of Magnitude include
    Date, ArithmeticValue, and Time, as well as Character and LookupKey.


Subclasses must implement the following messages:
    comparing
    <
    =
    hash'
```

```
!Magnitude methodsFor: 'comparing'

< aMagnitude
    "Answer whether the receiver is less than the argument."

    ^self subclassResponsibility


= aMagnitude
    "Answer whether the receiver is equal to the argument."

    ^self subclassResponsibility

hash
    "Answer a SmallInteger unique to the receiver."

    ^self subclassResponsibility
```

2.2.1 Reutilização de algoritmos: Métodos *Template*

As classes abstratas representam um conceito de grande importância do ponto de vista da reutilização de componentes. Do ponto de vista da semântica do domínio de aplicação uma classe abstrata representa uma abstração que define o comportamento genérico de um tipo de componente desse domínio. Sob esta interpretação uma classe abstrata define o comportamento comum de um conjunto de componentes, os quais se diferenciam no comportamento específico implementado para os métodos abstratos, enquanto a interface permanece constante.

A classe *Magnitude* apresenta um bom exemplo das capacidades de definição de componentes reutilizáveis através de classes abstratas. Em primeiro lugar a classe define as operações aplicáveis a qualquer magnitude, mas não a sua representação específica. Esta representação dependerá, obviamente, do tipo específico de magnitude, como por exemplo, *Integer*, *Float* ou *Date*. Também a classe fornece a implementação daqueles algoritmos que são independentes da representação, mas que podem depender da implementação específica dos métodos abstratos. Estes algoritmos, são denominados métodos *template*, pois definem uma estrutura de controle que chama operações cuja implementação é fornecida por subclasses ou outros objetos. Por exemplo, *Magnitude* fornece a implementação das operações de comparação que podem ser definidas em termos das operações abstratas, = e <:

```
<= aMagnitude
    "Answer whether the receiver is less than or equal to the argument."

    ^(self > aMagnitude) not

> aMagnitude
    "Answer whether the receiver is greater than the argument."

    ^aMagnitude < self

>= aMagnitude
    "Answer whether the receiver is greater than or equal to the argument."

    ^(self < aMagnitude) not

between: min and: max
    "Answer whether the receiver is less than or equal to the argument, max,
    and greater than or equal to the argument, min."

    ^self >= min and: [self <= max]
```

```
max: aMagnitude
    "Answer the receiver or the argument, whichever has the greater magnitude."

    self > aMagnitude
    ifTrue: [^self]
    ifFalse: [^aMagnitude]

min: aMagnitude
    "Answer the receiver or the argument, whichever has the lesser magnitude."

    self < aMagnitude
    ifTrue: [^self]
            ifFalse: [^aMagnitude].
```

O comportamento final de um método template varia em função da implementação dos métodos abstratos que ele chama, de acordo com a classe da instância que recebe uma mensagem que ativa esse método template. Por exemplo, uma classe File define a operação geral de cópia de um arquivo em outro pelo método *copyFile*, encarregado de fazer a cópia de dois arquivos, independentemente de que um arquivo seja local e o outro remoto.

```
class File {
  public:
      // Operações abstratas, dependenm de cada tipo de arquivo
      pure virtual Open;
      pure virtual Close;
      pure virtual Create;
      pure virtual Read;
      pure virtual Write;
                .....
      virtual copyFile();
}

File::copyFile(fileIn,fileOut)
File fileIn, fileOut;
{
      char buffer[255];

      fileIn.Open;
      fileOut.Create;
      while(fileIn.Read(buffer))
                fileOut.Write(buffer);
      fileIn.Close;
      fileOut.Close;
}
```

As operações de acesso (*read*, *write*, etc.) dependem do tipo de arquivo e deverão ser definidas por subclasses *LocalFile* e *RemoteFile* por exemplo, mas a operação de cópia é comum a ambos tipos de arquivos.

Do ponto de vista da codificação do comportamento de um domínio de aplicação, os métodos *template* representam, talvez, o conteúdo chave de uma classe abstrata. Através de métodos template é possível implementar o comportamento genérico das entidades de um domínio, o qual pode ser reutilizado sem mudanças em diferentes classes concretas. Este comportamento é adaptado através da redefinição dos métodos abstratos que o algoritmo chama. Deste modo, eles oferecem um mecanismo estruturado para codificar conhecimento acerca do comportamento de um domínio, permitindo a reutilização simultânea de interface e implementação.

2.2.2 Extensão de funcionalidade e reutilização de implementação

Freqüentemente as classes abstratas definem o comportamento de abstrações de muito alto nível, as quais precisam ser completadas por subclasses que representam abstrações de menor nível

ou de especificidade maior. A classe *Magnitude* apresenta, outra vez, um bom exemplo deste caso. *Magnitude* representa uma abstração que engloba tanto números puros como datas e caracteres. Assim, o protocolo geral definido por *Magnitude* é estendido por subclasses como *Number* e *Date*, as quais definem operações especificas à abstração que representam. Por exemplo, a classe *Number* é uma subclasse abstrata que define a interface para as operações aritméticas, +, -, *, etc. Subclasses de *Number* proverão implementações específicas para cada uma destas operações, dependendo da representação interna de cada tipo de número.

A classe *Date*, por sua vez, pode definir a interface para recuperar, por exemplo, o número do mês da data. Neste caso, a implementação deste método pode ser fornecida diretamente pela classe Date e não deveria ser redefinida por subclasses pois seu comportamento não admite variantes. Este tipo de métodos é denominado métodos ***base*** pois fornecem uma implementação completa de um comportamento comum a qualquer aplicação do domínio de aplicação.

Em outras circunstâncias, um classe pode fornecer uma implementação por falta, como poderia ser o nome do mês de uma data. Esta implementação por falta pode retornar o nome do mês em inglês, mas uma aplicação que precisa deste nome em português poderá por exemplo criar uma subclasse de Date que redefina esse método para retornar o nome em português. Este tipo de métodos são denominados métodos ***hook*** pois fornecem uma implementação de um comportamento de utilidade que, em certos casos, pode ser estendido por alguma subclasse. Isto é comum na seqüência de inicialização de componentes. Por exemplo, uma classe abstrata *View* pode definir o comportamento geral de objetos que realizam apresentações gráficas (janelas, ícones, etc.). Esta classe define uma visão como uma região retangular da tela e fornece o método *extent* para especificar a posição e tamanho da visão:

```
View:: bounds(x,y,height,width){
                originX = x;
                originY = y;
                extentX = height;
                extentY = width;
        }
```

Existem casos nos quais uma visão pode conter outras visões. Por isto, cada visão particular deverá distribuir o espaço disponível entre suas subvisões, segundo critérios que dependem de cada tipo de apresentação. Assim, cada subclasse de View que tenha subvisões deverá reimplementar o método *extent* para atribuir o espaço para suas subvisões, mas utilizando a implementação herdada para fixar o seu próprio tamanho:

```
EspecificView::extent(x,y, height,width){
                // Chama a implementação por falta
                View::extent(x,y, height,width);

                // Calcula posição e tamanho das subvisões
                subview.extent(sx,sy, sheight,swidth);
        }
```

Os métodos *hook* são outra ferramenta essencial provida por classes abstratas, para estender ou adaptar, de uma maneira estruturada, a funcionalidade de uma abstração. Através da definição adequada de métodos hook e template, é possível implementar uma estrutura adaptável que respeite protocolos estabelecidos e permita reutilizar tanto implementação quanto interface através de herança ou composição de objetos.

2.3 Frameworks

Uma aplicação orientada a objetos é construída com classes que colaboram, através da troca de mensagens, para realizar as tarefas do sistema. A distribuição de responsabilidades entre essas classes, e os padrões de colaboração entre elas, constituem o *projeto* (*design*)dessa aplicação.

Através dos mecanismos fornecidos pelo paradigma é possível reutilizar uma aplicação tanto como uma caixa-preta (através de uma interface que permita acessar os serviços que implementa)

quanto como uma caixa-branca redefinindo o comportamento de algumas subclasses. Assim podem se obter diferentes aplicações utilizando como base uma aplicação existente, reutilizando tanto o código como o projeto geral dessa aplicação. A quantidade de comportamento redefinido dependerá, evidentemente, do grau de semelhança entre as aplicações. Se as aplicações forem semelhantes, provavelmente só alguns métodos devam ser redefinidos e poucas classes serem especializadas, obtendo, em conseqüência, uma grande reutilização. Ao contrário, se as aplicações diferem muito, provavelmente muitas classes precisam ser redefinidas, diminuindo o ganho na reutilização.

No entanto, analisando posteriormente o comportamento redefinido é possível detectar que na realidade existe um parte de comportamento que é comum e que pode ser generalizado. Assim, o programador pode criar uma nova classe, superclasse das anteriores, que contenha todo o código comum a ambas, enquanto as originais implementam a porção específica do comportamento abstrato. Isto quer dizer, generalizando a partir da fatoração da comunalidade de vários casos concretos é possível obter uma classe abstrata que defina o comportamento genérico dessa família de casos, através de métodos abstratos, template, base e hook. Após este processo, a criação de uma nova aplicação, provavelmente, requererá a redefinição de muito menos comportamento que na situação anterior.

As classes abstratas representam conceitos genéricos relativos a uma família de entidades relacionadas. Cada entidade representa um caso particular da abstração é será representada por uma subclasse concreta. Esta subclasse fornecerá uma variante específica do comportamento abstrato definido na classe abstrata. Como já foi exposto, as classes abstratas funcionam como um molde para as suas subclasses. Da mesma forma, um projeto constituído por classes abstratas funciona como um *molde para aplicações*. Um projeto constituído por classes abstratas é denominado um *framework de aplicação orientado a objetos* ou simplesmente *framework* [DEU 89][JOH 88].

Informalmente um framework pode ser considerado como uma infra-estrutura de classes que fornecem o comportamento necessário para implementar aplicações dentro de um dado domínio, através dos mecanismos de especialização e composição de objetos, típicos das linguagens orientadas a objetos.

Uma aplicação construída com um framework pode ser conceitualizada como composta de dois níveis (FIGURA 2.1). Um nível superior que fornece a estrutura de controle da aplicação, constituído pelas classes do framework, e um nível inferior, constituído por subclasses concretas implementadas pelo usuário. Este nível fornece a implementação de operações específicas cuja ativação é modelada no nível superior. No caso geral, estas operações podem especializar a estrutura de controle de acordo com os requisitos da aplicação específica, bem como, chamar operações definidas no nível superior.

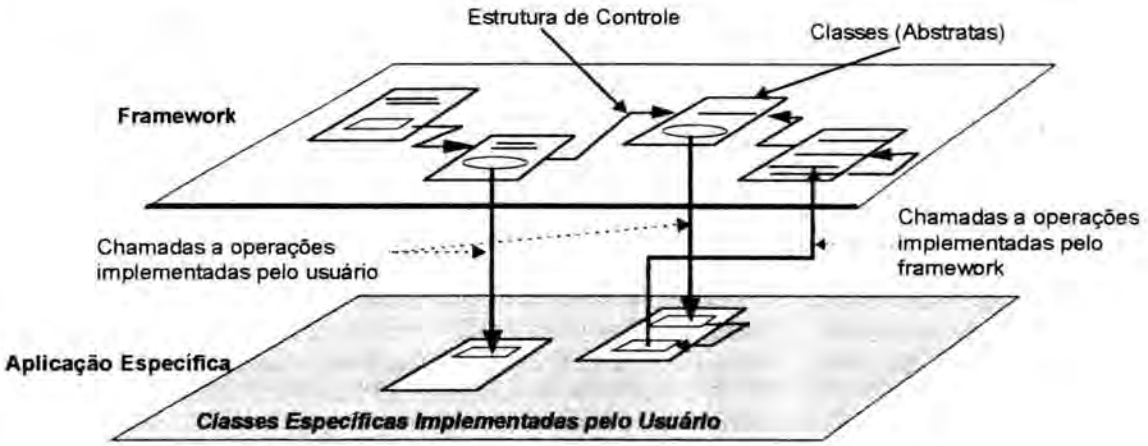


FIGURA 2.1- Visão conceitual da estrutura de um framework

De acordo com o estilo de projeto predominante que oferecem para construir aplicações, podem se diferenciar dois tipos de frameworks:

- ***Frameworks Baseados em Herança***: Os frameworks desta categoria, também denominados frameworks dirigidos pela arquitetura (*architecture-driven frameworks*) representam o caso mais comum. Estes frameworks fornecem a estrutura completa de uma aplicação (ou subsistema), a qual é completada com o código específico de cada aplicação particular. Eles são utilizados pela especialização de suas classes para implementar comportamento. Originalmente, foram denominados frameworks caixa branca devido à necessidade de conhecer-se seu funcionamento interno para sua adaptação [JOH 88].
- ***Frameworks Baseados em Composição***: Os frameworks desta categoria, também denominados dirigidos pelos dados (*data-driven frameworks*) ou frameworks caixa preta (*black box frameworks*), se caraterizam por serem utilizados através de composição de objetos para sua adaptação. Clientes adaptam o comportamento do framework utilizando diferentes combinações de objetos. O framework varia seu comportamento em função dos objetos que o cliente passa como parâmetros, mas o framework determina quais são as combinações válidas. Este tipo de frameworks constitui o basamento para a construção de *toolkits* de componentes.

Ambos os tipos de frameworks não são independentes um do outro. No caso geral frameworks baseados em composição são o resultado de sucessivas generalizações de um framework baseado em herança. Desenvolver desde o início frameworks baseados em composição é uma tarefa muito difícil e custosa [JOH 93]. No ciclo normal de desenvolvimento, um framework surge como um conjunto de classes obtidas através da generalização de vários exemplos. Em sucessivas tentativas de reutilização do framework, novas classes são criadas para implementar comportamentos não providos. Assim, após o desenvolvimento de várias aplicações, surgem novas oportunidades para refatorar o framework e generalizar ainda mais o comportamento das aplicações do domínio [OPD 92]. Deste modo, um framework que inicialmente fora concebido para ser utilizado por especialização, pode evoluir até uma estrutura paramétrica que pode ser instanciada através da combinação de objetos básicos que representam, cada um deles, diferentes funcionalidades do domínio.

Entretanto, estruturas desta natureza só podem ser facilmente obtidas em domínios estáveis e restritos, como por exemplo, interfaces com o usuário baseadas em diálogos. Um bom exemplo deste tipo de frameworks é representado pelo framework MVC descrito a seguir.

2.3.1 Um exemplo: O Framework MVC

O framework MVC, provido pelo ambiente *Smalltalk-VisualWorks* [PAR 94], para construção de interfaces com o usuário, representa um dos exemplos mais relevantes, existentes na atualidade, de estrutura de projeto adaptável dinamicamente, e de fatorização de abstrações que maximizam a reutilizabilidade de funcionalidade por composição de objetos. O grau de padronização de interfaces e encapsulamento de funcionalidade permite a construção de interfaces de diálogos com um editor de composição visual, através do qual, um programador pode descrever a composição da interface sem necessidade de programação.

A estrutura básica do framework enfatiza a separação entre a representação interna que a aplicação utiliza para conter os dados (Modelo), a sua apresentação visual (Visão) e o modo de acesso às funções de manipulação (Controlador). Conceitualmente, uma instância de uma visão tem sempre associado um objeto que representa o seu modelo e, opcionalmente, uma instância de um controlador.

O controlador coordena a atividade da sua visão, o modelo correspondente e ações do usuário através dos dispositivos de entrada. As visões representam a interface de saída da aplicação. Uma visão monitora as mudanças produzidas no seu modelo através de anúncios de mudanças que o modelo gera quando se produz alguma modificação de interesse no seu estado interno. A visão atualiza a apresentação na tela para manter a consistência entre o estado interno e o estado apresentado. Um modelo pode ter diferentes visões associadas, mas uma visão pode ter só um modelo associado.

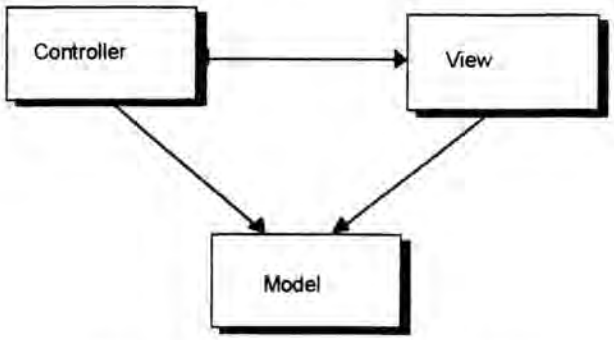


FIGURA 2.2. Estrutura Global do MVC

O framework promove a independência dos modelos das suas visões, através de um mecanismo indireto de anúncio de mudanças, de tal modo, que um modelo não possui conhecimento acerca das visões que possa ter associadas. Com este mecanismo uma visão se alista como dependente do seu modelo, através da mensagem *addDependent*. Quando no modelo se produz alguma mudança de estado, anuncia-se esta mudança através da mensagem *changed: < identificador de mudança> with: <argumentos>*. Esta mensagem fará que a visão dependente seja informada da mudança através da mensagem *update:with:*, a qual deve ser implementada para cada visão particular.

Assim, para prover uma aplicação com uma interface visual, um usuário do MVC deve compreender, em um primeiro lugar, esta estrutura básica para começar a utilizá-lo.

Um segundo aspecto é como construir esta interface gráfica utilizando as abstrações visuais providas. A FIGURA 2.3 apresenta uma visão parcial da hierarquia de classes visuais que compõem o framework. Esta hierarquia fatora a abstração básica de componente visual, representada pela classe abstrata *VisualComponent*, em um conjunto de classes concretas que definem visões com diferentes capacidades específicas. Cada tipo de visão da interface a ser construída, deverá ser implementada como subclasse de alguma destas classes concretas. Assim, também é necessário compreender a hierarquia de componentes concretos para determinar qual deles representa a abstração necessária para cada caso. A seguir descrevem-se sinteticamente as carateristicas mais relevantes desta hierarquia.

```
Object ()
    VisualComponent ()
        VisualPart ('container')
                CompositePart ('components' 'preferredBounds')
                        DependentComposite ('model')
                                CompositeView ('controller').....
                DependentPart ('model')....
                        View ('controller')
                                AutoScrollingView ('scrollOffset')
                ...
                Wrapper ('component')
                        GeometricWrapper ()
                                FillingWrapper ()
                                StrokingWrapper ('lineWidth')
                        GraphicsAttributesWrapper ('attributes')
                        TranslatingWrapper ('origin')
                                LayoutWrapper ('layout')
                                        BoundedWrapper ('extent')
                                                ....
```

FIGURA 2.3- Hierarquia Parcial de Classes Visuais do Ambiente Smalltalk

VisualComponent é a classe abstrata que define o comportamento genérico de objetos que geram representações visuais de si próprios. A abstração básica representada por *VisualComponent*

é a de uma região retangular dentro da qual as representações gráficas são geradas. Pode-se imaginar a composição de uma visualização como uma sobreposição de transparências de acetato, sendo que cada uma das quais provê uma área de desenho com seu próprio sistema de coordenadas

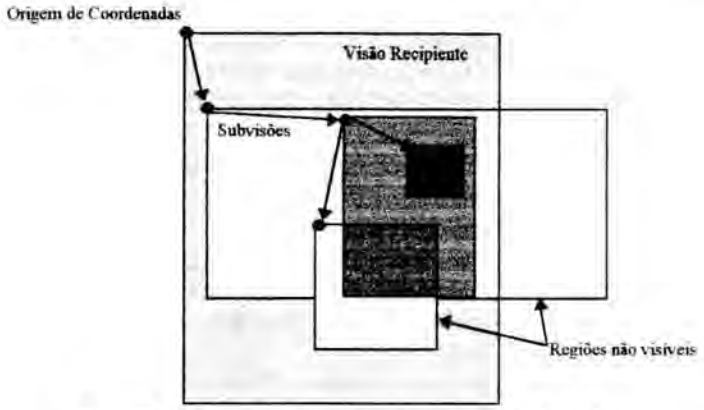


FIGURA 2.4- Organização de áreas de desenho

Subclasses implementam algoritmos particulares para produzir representações visuais de diferentes tipos de informação, fornecendo basicamente a implementação de dois métodos abstratos: *preferredBounds* para responder o tamanho da área que a visão deseja, e *displayOn:* para gerar a representação gráfica específica. Várias subclasses estendem a abstração básica provida por *VisualComponent* para fornecer mecanismos gerais de estruturação. A classe *VisualPart* estende *VisualComponent* para implementar visões estruturadas hierarquicamente, definindo um ponteiro ao componente que é o seu recipiente (variável *container*). Instâncias de *VisualPart* utilizam o seu recipiente para responder sua área. O recipiente age como um canal através do qual propagam-se dinamicamente requisições de atualização da apresentação gráfica quando uma *VisualPart* tem mudado seu estado. O recipiente é usualmente uma instância de *Wrapper*.

A classe *Wrapper* é projetada para ser recipiente de componentes visuais, aos quais transfere mensagens. Subclasses de *Wrapper* acrescentam tanto funcionalidade quanto propriedades visuais aos componentes visuais, como por exemplo, translação, recorte, bordas, etc. A subclasse *TranslatingWrapper*, por exemplo, especializa *Wrapper* para fornecer a funcionalidade de translação de coordenadas de componentes visuais. Assim, uma visão sempre define seu origem de coordenadas no ponto (0,0), sendo o wrapper o encarregado de aplicar a translação até o ponto real de origem dentro do sistema de coordenadas definido pela visão pai.

A Classe *CompositePart* estende *VisualPart* para fornecer o suporte para implementar visões compostas através de uma coleção que contém a subestrutura de visões (variável de instância *components*). Os elementos desta coleção assumem-se serem *wrappers* que contêm componentes visuais, o qual permite a composição recursiva de objetos compostos. Assim, a estrutura típica de uma visualização construída utilizando o framework será uma árvore composta por camadas alternadas de componentes visuais e wrappers.

O mecanismo de associação de *wrappers* é, geralmente, invisível para o usuário do framework. Quando um componente visual é adicionado à estrutura de uma visão composta, o mecanismo de adição solicita à classe do componente visual qual a classe de wrapper de translação que deve ser utilizada para criar o wrapper associado, através da mensagem *translatingWrapperClass*. A implementação por falta desta mensagem retorna a classe de translação padrão *TranslatingWrapper*.

As classes *DependentPart* e *DependentComposite* acrescentam a capacidade de que uma visão simples ou composta, respectivamente, tenha associado um modelo, através da variável de instância *model* e os métodos associados para seu acesso. A classe *View* e *CompositeView*, por sua vez, acrescentam o comportamento e a estrutura necessários para que uma visão simples ou composta, respectivamente, possuam um controlador associado.

Como é simples observar, para programar cada novo componente visual é necessário tomar em consideração não só os métodos que devem ser implementados, mas a organização de instâncias e convenções que o framework predefine.

Dependendo da estrutura da visualização, a área real ocupada por uma visão pode ser determinada por uma outra visão que a contém. Os clientes de um componente visual requisitam a região ocupada por ele através da mensagem *bounds* e o informam da área que ocupará através da mensagem *bounds: anArea*. O redimensionamento de um componente implicará no envio da mensagens *bounds:* com os novos limites (origem, altura e largura) onde será desenhado. Se a visão tem associada uma instância de *TranslatingWrapper*, o ponto de origem será mudado à nova origem, o componente será informado de seu novo tamanho e será propagado até o topo da hierarquia um anúncio de mudança que provocará a atualização de toda visualização através da mensagens *displayOn:*. O mecanismo de recorte (*clipping*) faz com que só as partes da visualização que foram afetadas pela mudança recebam esta mensagem.

A fatoração de funcionalidade favorece muito a reutilização de comportamentos ortogonais através da composição de objetos. Diferentes visões podem ser deslocadas, possuir bordas, mudar de tamanho, etc., através da composição de wrappers. Além disso, estas propriedades podem ser mudadas dinamicamente trocando o tipo de wrapper que uma visão particular tem associado, fornecendo, deste modo, um alto nível de flexibilidade para construir interfaces com o usuário adaptáveis dinamicamente.

2.4 Frameworks, Modelos e Arquiteturas

Na seção precedente apresentou-se informalmente o conceito de framework como uma infra-estrutura de classes que fornece o suporte reutilizável para construir aplicações dentro de um dado domínio de aplicação. Na literatura encontram-se múltiplas definições semelhantes de framework, as quais colocam a ênfase em aspectos parciais das suas caraterísticas.

Johnson destaca os aspectos abstratos do projeto como o aspecto central que carateriza os frameworks.

- *"Um framework é constituído por um conjunto de classes abstratas e componentes que em conjunto definem um projeto abstrato para uma família de problemas relacionados"* [JOH 92].
- *"Um framework é um conjunto de classes cooperantes que constituem um projeto reutilizável para uma classe específica de software..... O framework define a arquitetura da aplicação. Ele define a estrutura global da aplicação, a sua divisão em classes e objetos, as responsabilidades chave de cada parte, como as classes e os objetos que colaboram e a seqüência de controle."* [GAM 94]

Pree ressalta os aspectos estruturais de classes e de sua utilização por especialização para a construção de aplicação[PRE 94]:

- *"Um framework é constituído por um conjunto de blocos de construção prontos para serem utilizados e outros semi-acabados. A arquitetura global, isto é, a composição e interação de blocos, é predefinida também. Produzir uma aplicação específica usualmente envolve a adaptação de componentes a necessidades específicas implementando alguns métodos em subclasses das classes do framework"*

Goldberg, por sua vez, centra-se nos aspectos operacionais, ou dinâmicos, de um framework, tomando em consideração o aspecto da estrutura de controle que esse framework fornece para a construção de aplicações [GOL 95]:

- *"Um framework é um conjunto de objetos que interagem que fornecem um conjunto bem definido de serviços, e uma forma bem definida na qual o controle é transferido entre esses objetos".*

Deutsch, por sua vez, define o conceito de framework tomando em consideração o contexto de utilização das abstrações e de extensão dessas abstrações [DEU 89]:

- *"Um cliente pode prover uma instância X de uma classe própria como parâmetro a uma instância Y de uma outra classe existente, com a expectativa de que Y enviará um conjunto de mensagens preestabelecidos a X. Neste caso Y é considerado como o framework enquanto X é considerado um cliente interno para diferenciá-lo dos clientes externos que interagem com a combinação."*
- *"Um cliente pode definir uma subclasse XC de uma classe existente YC, fornecendo funcionalidade adicional ou especializada. Outra vez, a classe existente é considerada como o framework e o cliente um cliente interno".*

Estas definições, colocam a ênfase em diferentes aspectos ou perspectivas tanto da estrutura como da natureza operacional dos frameworks. A definição de Deutsch, é talvez a mais precisa em termos dos mecanismos que a orientação a objetos oferece para suportar reutilização de funcionalidade. Esta definição, entretanto não destaca os aspectos de *abstração* inerentes aos frameworks, nem os aspectos de *domínio de aplicação*, fazendo referência implícita à predefinição de uma estrutura de controle. Este aspecto é ressaltado pela definição de Goldberg, mas, neste caso, o aspecto de abstração está implícito na definição. Pree, e também Gamma, Helm, Johnson e Vlissides, incluem o conceito de *arquitetura* como um dos aspectos essenciais que caraterizam um framework, o qual está intimamente relacionado com a noção de estrutura de controle bem definida referida por Goldberg.

O conceito de domínio de aplicação está relacionado com o objetivo de um framework, o qual é o fornecimento de uma estrutura reutilizável para produzir aplicações semelhantes. A abstração carateriza o processo de construção de um framework, enquanto o conceito de arquitetura carateriza o resultado deste processo.

2.4.1 Modelo de domínio de aplicação

Um *domínio de aplicação* pode ser definido como um conjunto de aplicações existentes e futuras as quais possuem caraterísticas semelhantes. Estas caraterísticas semelhantes estão constituídas por entidades, operações, eventos e relacionamentos que *abstraem* os aspectos comuns ou regulares desse domínio [ARA 94].

A descrição e classificação destas abstrações, através de alguma linguagem (possivelmente formal), define um *modelo* para esse domínio.

A construção de um modelo de domínio é o resultado de um processo de *análise de domínio* [ARA 91]. Através deste processo, aplicações existentes e outras fontes de conhecimento do domínio são analisadas, na procura das *abstrações* que representam os componentes genéricos desse domínio. Um modelo de domínio identifica, em termos abstratos, as partes constituintes e os relacionamentos entre elas, mas idealmente, não aprofunda na constituição interna nem na forma em que esses relacionamentos se materializam em termos de uma implementação computacional.

2.4.2 Arquitetura de software

O termo *arquitetura de software* é utilizado em diferentes contextos para descrever diferentes aspectos de sistemas de software, desde a descrição do seu projeto detalhado até a descrição de um estilo que define como as partes constituintes de um sistema interagem em um alto nível de abstração. Freqüentemente os termos arquitetura e modelo são tratados como sinônimos ou utilizados

alternativamente para denotar a estrutura global de um sistema de software. Entretanto, é conveniente realizar uma distinção entre ambos conceitos.

Um modelo é um dispositivo analítico através do qual propriedades relevantes de um artefato podem ser estudadas e formalizadas [COC 91]. Um modelo é caraterizado pela abstração. Um modelo deve, idealmente, refletir aquelas propriedades relevantes do artefato ou realidade a serem estudadas, omitindo todo detalhe não relevante para tal estudo. Sob esta interpretação, um mesmo artefato pode admitir diferentes modelos, dependendo do objetivo da modelagem.

Arquitetura de software é definida por Abowd, Allen e Garlan como um nível de descrição de sistema, no qual decisões de projeto chave são expressadas, como a divisão de granularidade grossa de um sistema em subsistemas que interagem, a atribuição de funcionalidade a componentes computacionais, protocolos de interação entre estes componentes, propriedades computacionais globais do sistema (como latência e desempenho) e propriedades de qualidade como manutenibilidade e reutilizabilidade [ABO 95].

A arquitetura de um sistema de software reflete um conjunto de decisões de projeto que respondem a restrições computacionais da solução do problema modelado. Sendo uma descrição de um sistema, a arquitetura representa um modelo desse sistema, no qual as *propriedades computacionais* dos componentes, as *interfaces* e *fluxo de controle* entre eles são explicitamente definidas e descritas. Assim, enquanto um modelo de um sistema *descreve* as propriedades e comportamento desse sistema, a arquitetura representa um refinamento, ou materialização, dos conceitos abstratos definidos por esse modelo, a qual *prescreve* uma solução computacional.

No nível de arquitetura, a forma em que os componentes são efetivamente implementados não é fixada, mas sim a forma em que as entidades e relacionamentos definidos pelo modelo são mapeados a componentes que implementarão o comportamento do sistema. A natureza deste mapeamento depende das restrições do entorno de implementação e as propriedades que o software resultante da implementação deva possuir (extensibilidade, eficiência, manutenibilidade, etc.).

Quando uma arquitetura baseia-se em um modelo de domínio, tal como definido acima, então essa arquitetura representa uma arquitetura específica para esse domínio. Uma arquitetura específica para um domínio captura a estrutura genérica das aplicações do domínio, fornecendo a infra-estrutura que habilita a construção de aplicações através de instanciação de componentes específicos.

A utilização de arquiteturas genéricas para o desenvolvimento de software constitui uma das formas mais poderosas de reutilização: a reutilização de projeto [KRU 92]. O processo inicial de projeto consiste em particionar um dado problema em um conjunto de componentes que colaboram para resolver esse problema. Neste ponto, um modelo pode ajudar no processo de particionamento, enquanto uma arquitetura reutilizável, adequada para o problema, pode evitar este processo. Uma arquitetura genérica desenvolvida por especialistas em um domínio, pode induzir uma estrutura adequada para uma família de aplicações, guiando, portanto, o processo de projeto. Uma estrutura adequada reduz as possibilidades de um projeto inflexível, permitindo que os projetistas se concentrem nos aspectos importantes da aplicação particular sendo desenvolvida[COC 91].

2.4.3 Frameworks: Arquiteturas genéricas orientadas a objetos

Uma arquitetura genérica para um domínio de aplicação pode ser implementada através de uma linguagem orientada a objetos, na qual classes abstratas representam os componentes genéricos da arquitetura e métodos template definem a estrutura de controle genérica das aplicações. Isto é, o resultado desta implementação será um framework para esse domínio de aplicação. A implementação particular de cada componente dependerá, obviamente, das capacidades fornecidas pela linguagem. Linguagens dinâmicas, como CLOS e Smalltalk, favorecem a implementação de estruturas altamente flexíveis e o mapeamento direto de construções de projeto. Linguagens tipadas impõem limitações que podem forçar implementações diferentes da mesma arquitetura.

Portanto, tomando em consideração estes aspectos, um framework pode ser definido como:

> *"Um framework é uma implementação, em uma linguagem orientada a objetos particular, de uma arquitetura específica para um modelo de domínio de aplicação, em termos de classes abstratas e componentes."*

Esta definição toma em consideração os aspectos centrais que caraterizam o processo de construção de frameworks. O modelo de domínio carateriza o processo de abstração de caraterísticas comuns de um domínio de aplicação, o qual é inerente aos frameworks. A arquitetura define a estrutura computacional do domínio, refletindo decisões específicas de projeto que tratam sobre aspectos da distribuição do processo de computação entre classes e objetos, que não tem todos, necessariamente, uma correspondência direta com as entidades definidas pelo modelo de domínio. Finalmente, um framework é programado em uma linguagem específica, podendo variar em muito a implementação da arquitetura de uma linguagem para outra.

2.4.4 Estilos arquitetônicos e projeto de frameworks

A importância da arquitetura dos sistema de software motivou uma ativa pesquisa nos últimos anos, com o objetivo de fornecer mecanismos e notações formais para a sua descrição. Uma linha de trabalho muito importante, que deve ser destacada, é representada pelos trabalhos de Garlan e Shaw sobre *estilos arquitetônicos* [SHA 96][GAR 94][ABO 95]. Um estilo arquitetônico descreve uma forma particular na qual os diferentes componentes de nível global de um sistema transferem dados e controle entre eles. Exemplos clássicos destes estilos são as comumente denominadas arquitetura de camadas e arquitetura cliente-servidor, entre outros. Estes estilos, entretanto, representam uma descrição abstrata da forma em que os componentes interagem, ou seja, representam, na realidade, um modelo, cujo universo de discurso são componentes de software. Já a atribuição de funcionalidade entre componentes e a definição de suas interfaces representam uma arquitetura que corresponde com esse estilo arquitetônico.

Por exemplo, o estilo *pipeline* (mais especificamente *pipes-and-filters* [SHA 96]) carateriza os domínios cujas aplicações podem ser decompostas em sub-tarefas independentes de transformação de dados. Compiladores e sistemas de graficação, 2D e 3D, correspondem a este modelo. No primeiro caso, o processo é dividido em fases consecutivas de análise léxica, sintática, semântica e geração de código. Diferentes compiladores podem fornecer diferentes funcionalidades que acrescentam componentes ao *pipeline*, como por exemplo otimizadores de código, os quais podem ser aplicados opcionalmente. Uma arquitetura genérica para o domínio dos compiladores respeitará este modelo, fornecendo os mecanismos genéricos que caraterizam cada tipo de componente. No segundo caso, o processo de produzir um gráfico na tela é constituído por tarefas como conversão de sistemas de coordenadas, projeção, recorte e desenho. Uma arquitetura genérica para este domínio também respeitará a estrutura descrita pelo estilo, mas dificilmente poderá ser reutilizada para construir um compilador.

Do ponto de vista do projeto de uma arquitetura genérica para um domínio de aplicação, e, em conseqüência, para o projeto de um framework, o reconhecimento de um modelo ou estilo arquitetônico para esse domínio representa um passo de grande importância. Sistemas complexos são habitualmente compostos por subsistemas que encapsulam funções específicas e fornecem serviços aos outros subsistemas, de acordo com algum padrão de comunicação determinado pela arquitetura. Quando um estilo arquitetônico é reconhecido para um domínio, o processo inicial de projeto consiste em distribuir funcionalidade entre os componentes descritos pelo estilo. Cada subsistema ou componente, por sua vez, pode ser implementado por uma classe que define a interface de acesso aos seus serviços, enquanto a implementação específica desses serviços pode ser distribuída entre um outro conjunto de classes.

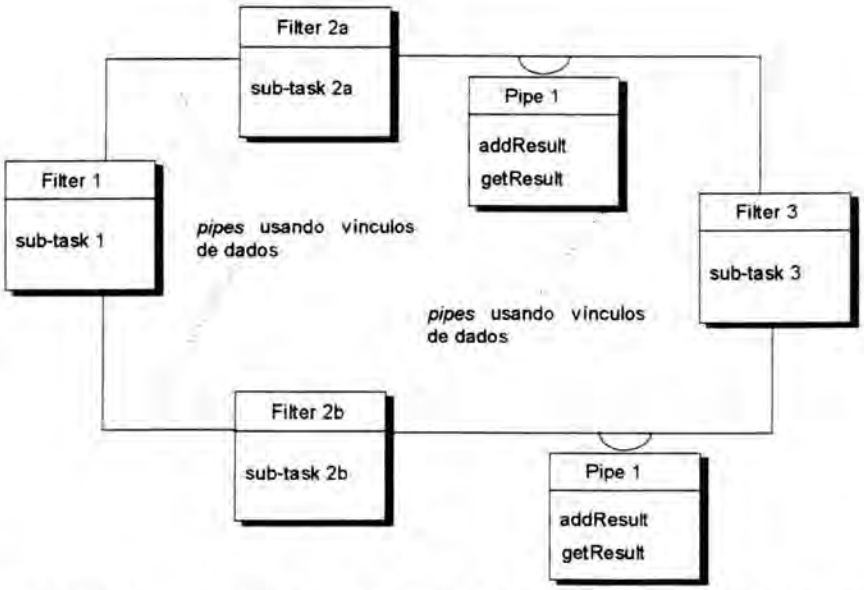


FIGURA 2.5- Representação genérica do estilo pipes-and-filters através de classes

Com esta abordagem, Buschman. e Meuneir, mostram como a maior parte dos estilos arquitetônicos reconhecidos podem ser modelados em termos de objetos [BUS 96]. Neste caso, os estilos são denominados *architectural frameworks*, para distingui-los de estruturas menores de projeto e implementação como os *padrões de projeto* [GAM 94] e os *idiomas* [COP 92]. Por exemplo, a FIGURA 2-5, mostra a descrição genérica de um estilo *pipes-and-filters* utilizando o modelo estático de OMT [RUM 91].

Batory e O'Mailley utilizaram esta mesma abordagem para representar sistemas baseados em camadas, através de uma implementação orientada a objetos. Em sua abordagem cada camada é suportada por um framework, constituindo, deste modo, uma arquitetura genérica. Cada componente da arquitetura assume-se como definindo um tipo (*realm*) que oferece uma interface opaca para as outras camadas. Com a presunção de que o domínio corresponde a uma arquitetura de camadas, a estrutura de componentes de um sistema desse domínio pode ser expressada através de uma linguagem simples baseada em gramáticas. Através de expressões desta linguagem define-se um modelo do domínio que descreve a forma em que aplicações podem ser construídas através de combinações de componentes [BAT 92a] [BAT 92b].

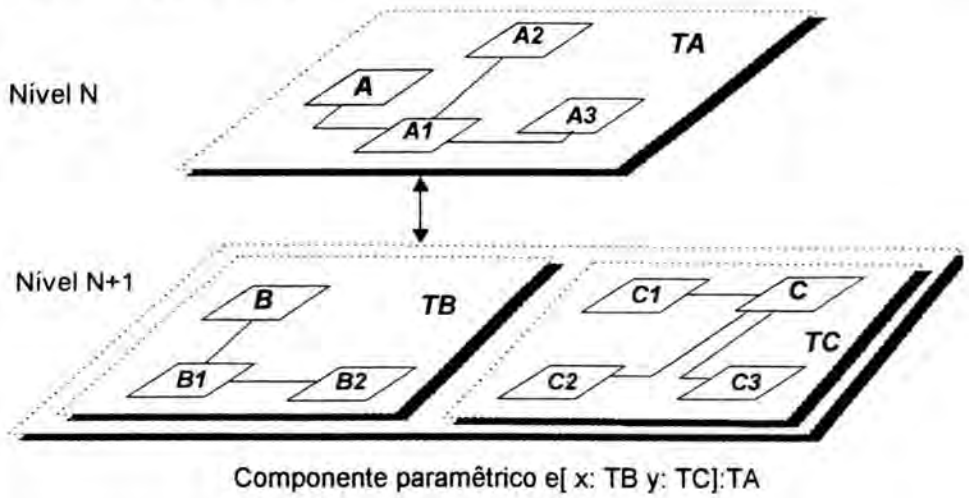


FIGURA 2.6- Representação de uma arquitetura de níveis composta por frameworks

Deste modo, um domínio complexo pode ser representado por um framework que implementa a arquitetura global das aplicações do domínio, definida por algum modelo arquitetônico. Cada componente da arquitetura pode ser, por sua vez, implementado por diferentes sub-frameworks, os quais provêm o suporte reutilizável para a construção de subsistemas especializados. O framework para sistemas operacionais Choices [JOH 91] é um exemplo deste tipo de projeto, o qual divide a estrutura de um sistema operacional em quatro componentes principais, cada um dos quais é materializado por um sub-framework. O framework Luthier, descrito no capítulo 6, também corresponde a este padrão.

2.5 Frameworks e Instanciação de Aplicações

A definição apresentada na seção anterior não responde diretamente qual a diferença entre um framework e uma hierarquia de classes, ou reciprocamente, quando uma hierarquia de classes pode ser considerada um framework. A resposta a estas questões está diretamente relacionada com a capacidade do framework para permitir construir aplicações dentro do domínio.

O processo de construção de uma aplicação específica do domínio de aplicação utilizando um framework é denominado *instanciação* [JOH 88]. Este processo consiste, geralmente, de duas fases [WIR 90]:

- ***Implementação do comportamento específico***: A forma habitual de utilizar um framework é através do mecanismo de herança, criando subclasses das classes abstratas que definem o framework. Estas subclasses deverão, basicamente, implementar o comportamento específico dos métodos abstratos definidos nas classes abstratas, além do comportamento adicional necessário para satisfazer os requisitos da aplicação.
- ***Composição de instâncias***: Quando todos os componentes necessários estão desenvolvidos, o usuário deve descrever como as instâncias destas classes se conectam para formar a aplicação.

Estas duas fases caracterizam o processo básico de construção de aplicações de frameworks baseados em herança. Quando todos os componentes estão prontos na biblioteca, uma aplicação pode ser construída simplesmente descrevendo como este componentes se combinam. Esta situação, entretanto, só é possível em domínios muito restritos, nos quais a quantidade de comportamentos diferentes é limitada. Em domínios mais amplos, é necessário, geralmente, a especialização de classes para fornecer o comportamento específico de cada aplicação ou subsistema. A questão essencial é quanto comportamento deve ser implementado e, essencialmente, que tipos de métodos devem ser redefinidos.

Potencialmente, qualquer projeto constituído por classes abstratas poderia ser considerado um framework, mas o fato desse projeto estar definido por classes abstratas não garante que seja totalmente reutilizável para construir aplicações dentro do seu domínio. Não necessariamente essas classes definem o conjunto adequado de abstrações, nem a interface adequada que permita, através da composição de objetos e a redefinição de alguns métodos, construir aplicações dentro do domínio.

Idealmente, uma hierarquia de classes constitui um framework quando é possível construir *qualquer* aplicação dentro do domínio, utilizando os recursos e a estrutura de controle definidos pelo framework. Isto é:

> *Um projeto $\mathcal{F}$ constituído por classes abstratas, é um framework para um domínio de aplicação $\mathcal{D}$, se qualquer aplicação do domínio pode ser gerada através da implementação de métodos abstratos e hook, definidos nas classes abstratas, e a utilização de componentes concretos existentes na biblioteca.*

Sob esta definição, $\mathcal{F}$ constitui um framework *ideal* para $\mathcal{D}$, pois implica que o framework fornece a funcionalidade e a estrutura de controle genérica que define os pontos de adaptação *necessários* e *suficientes* para satisfazer um conjunto, potencialmente infinito, de aplicações.

Esta definição toma em consideração o aspecto central que distingue os frameworks de simples hierarquias de classes. Um framework fornece uma *estrutura de controle para todas as aplicações do seu domínio*, cujas funcionalidades são concretizadas através da implementação dos pontos de extensão e adaptação previstos pelo framework, mantendo invariável o protocolo definido pelas classes abstratas. No caso ideal, é necessário tomar em consideração que uma aplicação particular poderia acrescentar métodos privados, os quais implementam funções específicas à aplicação. Mas, não seria necessário acrescentar novos métodos públicos, relacionados com funcionalidade do domínio de $\mathcal{F}$, que modifiquem ou alterem o fluxo de controle preestabelecido entre os componentes.

Garantir que um dado framework cumpre com estes requisitos, obviamente, não é possível praticamente. Em primeiro lugar, os problemas de reutilização surgem só quando tenta-se utilizar o framework para construir aplicações específicas, as quais não podem ser completamente previstas pelos projetistas. Também, é necessário tomar em consideração, que um framework é, habitualmente, o resultado de um processo iterativo de projeto, no qual os exemplos desenvolvidos tem um papel preponderante. Se a seleção destes exemplos não é suficientemente representativa do domínio, então o framework resultante, muito provavelmente, não fornecerá uma estrutura adequada. Isto implicará na necessidade de novas refatorações, na medida que novas aplicações sejam desenvolvidas.

Neste sentido, a definição poderia ser de utilidade para medir o grau de generalização, ou maturidade, provido por um dado framework. Se através de sucessivas tentativas de reutilização, as aplicações podem ser desenvolvidas fornecendo a implementação de métodos que foram projetados para serem redefinidos, acrescentando poucos métodos adicionais que aumentam a interface de um componente, então esse framework ofereceria uma infra-estrutura adequada para seu domínio. Se, ao contrário, é necessário acrescentar muitos métodos adicionais, então o framework não forneceria uma infra-estrutura o suficientemente madura para representar o domínio.

O grau de certeza desta medição, entretanto, depende do grau de compreensão que o programador tenha alcançado do framework. Isto é, as capacidades efetivas de um framework para produzir aplicações só podem ser realmente avaliadas, quando o usuário possui uma *compreensão completa* das funcionalidades que o framework oferece e dos protocolos definidos pelas suas classes abstratas. A dificuldade para alcançar este grau de compreensão é, precisamente, uns dos principais problemas que os frameworks apresentam.

3 Compreensão de Frameworks

A dificuldade da utilização de frameworks para construir aplicações é, talvez, o fator mais limitante da sua aplicabilidade para a produção de software. No capítulo precedente apresentou-se um exemplo simples da forma em que um framework é utilizado para construir aplicações, sem aprofundar nas dificuldades e problemas relacionados com esta utilização. Este capítulo objetiva apresentar uma análise detalhada dos aspectos que contribuem à dificuldade de compreensão através da introdução de um cenário real de uso de um framework complexo.

A seção 3.1 apresenta uma definição de compreensão de software em geral, discutindo os fatores que contribuem a tornar difícil essa compreensão e a complexidade agregada pelas linguagens orientadas a objetos. A seguir, a seção 3.2 apresenta um exemplo real de um problema relacionado com a utilização do framework MVC do ambiente *Smalltalk-VisualWorks* [PAR 94]. Na seção 3.3, analisam-se as limitações das técnicas de documentação providas pelos métodos de desenvolvimento mais difundidos e as técnicas especiais de documentação propostas na literatura. Finalmente, na seção 3.4, discutem-se o papel dos exemplos na compreensão de frameworks e a aplicação de técnicas de documentação eletrônica para apoio na utilização de frameworks.

3.1 Compreensão de Software

A *compreensão de programas* é um dos problemas mais críticos dentro do ciclo de vida de software. O sucesso de atividades tais como manutenção ou depuração dependem, em grande parte, da facilidade com a qual um programador possa compreender um dado programa quando surge a necessidade de modificar a sua funcionalidade ou corrigir erros. Este problema também afeta diretamente a facilidade com a qual um artefacto de software pode ser reutilizado. A manutenção e a reutilização podem ser vistos como facetas de uma mesma atividade [JOH 88]. A reutilização de artefatos de software é caraterizada pela necessidade de realizar mudanças ou adaptações ao artefato a ser reutilizado, para adequá-lo aos requisitos da nova aplicação [KRU 92]. Deutsch sugere que só existe *reutilização* de software quando existe alguma mudança, seja no artefato reutilizado ou no contexto no qual é utilizado. Assim, independentemente da tecnologia de reutilização utilizada (componentes de código fonte, esqueletos de software, linguagens de muito alto nível, etc.), um programador deve *compreender* a natureza das abstrações utilizadas para selecionar dentre os componentes disponíveis aqueles que satisfazem os requisitos da nova aplicação, e efetuar as modificações necessárias ou estendê-lo com funcionalidade específica [DEU 89].

Geralmente, entende-se por compreensão a capacidade ou conhecimento que possui uma pessoa para explicar um dado conceito ou artefato, tanto do ponto de vista do seu significado como do seu funcionamento. No caso de artefatos de software, pode considerar-se que uma pessoa compreende um dado artefato quando é capaz de explicar como ele funciona, e modifica-lo para adaptá-lo a novos requisitos, ou utilizá-lo adequadamente para o objetivo que foi projetado. Biggerstaf, Mitbander e Webster ressaltam o aspecto da associação de conceitos relativos ao domínio de aplicação com conceitos relativos ao domínio de programação [BIG 93]. Um programa expressa uma solução para um problema dentro de um domínio de aplicação em termos de uma linguagem formal, a qual constitui o domínio de programação. Uma pessoa compreende um dado programa, quando é capaz de explicar a sua estrutura, seu comportamento, seus efeitos sobre seu contexto, e seu relacionamentos com o domínio de aplicação, em termos de comunicação humana. Isto é, uma pessoa compreende um dado programa quando é capaz de explicar, em termos informais do domínio de aplicação, o processo computacional expressado pelos elementos sintáticos do programa que codificam a solução do problema.

Quando um programador enfrenta a necessidade de compreender um novo programa, deve reconstruir o mapeamento entre a expressão formal da solução dada pela linguagem de programação, aos conceitos informais do domínio de aplicação. Esta reconstrução implica,

habitualmente, em um *processo dedutivo* de aquisição de conhecimento acerca do artefato em consideração, que envolve atividades tais como análise, abstração e generalização de conceitos [TIL 96]. Através deste processo, o programador descobre gradualmente a associação entre os conceitos informais do domínio de aplicação com o domínio do programa. Tal processo de compreensão, porém, é uma tarefa reconhecidamente complexa e demorada.

3.1.1 Fatores limitantes

A dificuldade de compreensão de programas pode ser atribuída a uma combinação de fatores que abrangem aspectos tais como complexidade do domínio de aplicação, limitações cognitivas dos programadores, práticas de desenvolvimento pouco apropriadas e características da linguagem de programação.

A complexidade do domínio de aplicação é certamente um fator essencial que pode tornar complexa a sua solução computacional. Domínios altamente complexos e restritos, como por exemplo sistemas de controle em tempo real, ou sistemas distribuídos, podem alcançar várias centenas de milhares de linhas de código, com fortes dependências entre os diferentes componentes. Realizar uma mudança nestes sistemas implica, habitualmente, na necessidade de compreender o projeto global para localizar a porção de código a ser modificada e propagar as mudanças necessárias aos componentes dependentes. Scheinewind sugere que o problema de localizar quais partes do código de um sistema devem ser modificadas é o problema central da manutenção de software [SCH 87].

As metodologias de desenvolvimento e as linguagens de programação evoluíram durante as últimas duas décadas, em parte, para minimizar os problemas da manutenção do software. Conceitos como programação estruturada, modularidade e abstração de dados foram progressivamente incorporados como elementos fundamentais das linguagens de programação modernas, definindo, essencialmente, mecanismos de encapsulamento através de âmbitos sintáticos, que permitem controlar o impacto produzido por mudanças locais no código sobre o resto do sistema. Estes mecanismos contribuem, certamente, para a produção de código de maior qualidade em termos de legibilidade e manutenibilidade. Entretanto, centram-se mais nas características formais da expressão computacional da solução, do que em fornecer um suporte adequado para ser compreendido por um leitor interessado em determinar a funcionalidade de um programa ou parte dele [KNU 84].

Esta limitação poderia ser resolvida com documentação de projeto apropriada, mas, no caso geral, as metodologias de desenvolvimento mais difundidas, falham na provisão de formalismos adequados para refletir o mapeamento entre a implementação e o projeto [BAX 92][ARA 93]. Sistemas complexos se caracterizam por dependências que abrangem várias dimensões de projeto, como por exemplo funcionalidade, performance, requisitos de memória, interoperabilidade, etc. A natureza informal da documentação e seu relacionamento indireto com o código final, força os programadores a recuperar e compreender, através da análise de código, as soluções adotadas em cada dimensão, para propagar as mudanças necessárias. Baxter [BAX 92] e Arango [ARA 93] atribuem a falta de mecanismos que permitam refletir o *raciocínio* que conduziu a determinadas decisões de projeto como falha fundamental dos métodos de desenvolvimento. Esta informação não pode ser adequadamente refletida pelas construções sintáticas das linguagens de programação, pois depende de restrições de contexto que definem uma determinada solução computacional. Assim o programador deve deduzir estas restrições a partir de uma documentação que só reflete o processo computacional que resolve o problema, com o conseqüente risco de interpretações erradas que introduzem novas falhas no sistema.

As práticas de manutenção impostas pela necessidade de realizar modificações em tempos reduzidos, podem piorar ainda mais este problema. Freqüentemente o código é modificado e essas mudanças não são refletidas na documentação, nem mesmo os próprios comentários do programa são modificados de acordo com a mudança do código que documentam. Deste modo, a documentação fica desatualizada e o código fonte converte-se na única fonte de referência confiável.

Diante desta situação, os aspectos *cognitivos* convertem-se em outro dos fatores que influem fortemente na facilidade de compreensão. As limitações humanas para processar ou assimilar grandes quantidades de informação foram caracterizadas por Miller em seu já clássico artigo, *The*

Magical Number Seven, Plus or Minus Two: Limits of Our Capacity to Process Information [MIL 56]. Um pessoa é capaz de assimilar, simultaneamente, aproximadamente sete itens de informação, na memória denominada de *curto prazo*. Estes itens não são lembrados após menos de um minuto, se não são novamente exercitados. Um item é uma peça simples de informação, a qual pode representar um dado elementar ou uma estrutura complexa tratada como uma unidade. A memória de curto prazo domina o mecanismo de processo consciente de informação que habilita os seres humanos a compreender novos conceitos. Esta limitação é um dos principais problemas que enfrenta um programador para compreender e manipular sistemas complexos, pois inibe a capacidade de processar simultaneamente as diferentes partes do sistema que interagem entre si.

Para diminuir estas limitações, o programador deve agrupar informação básica em unidades de maior conteúdo semântico, as quais podem ser posteriormente utilizadas para relacioná-las com outras partes do sistema. Curtis considera este processo de agrupamento de informação como um dos mecanismos fundamentais da compreensão e reutilização de artefatos software [CUR 89]. Programadores experientes se caraterizam por possuir um maior conhecimento de estruturas abstratas de programação, as quais são recuperadas rapidamente da denominada *memória de longo prazo*, na presença de itens que correspondem com essas abstrações na memória de curto prazo. Deste modo, o conhecimento de padrões preestabelecidos facilita a compreensão imediata de unidades funcionais, sem requerer a análise detalhada de cada sentença. Um exemplo tipico destes padrões é o processo da linha de comandos em programas C. A estrutura habitualmente utilizada para o reconhecimento de comandos tipo *Unix*, corresponde aproximadamente com uma estrutura da forma:

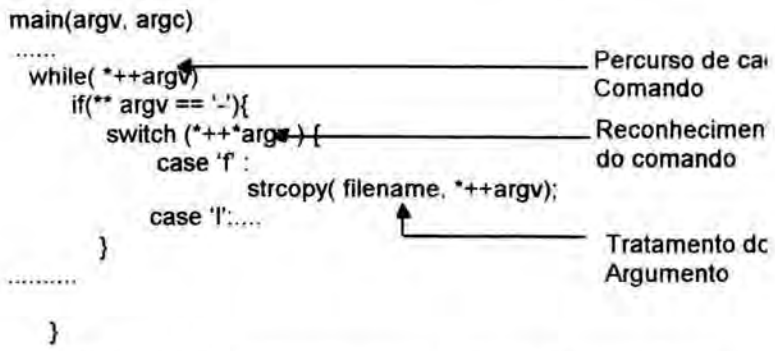


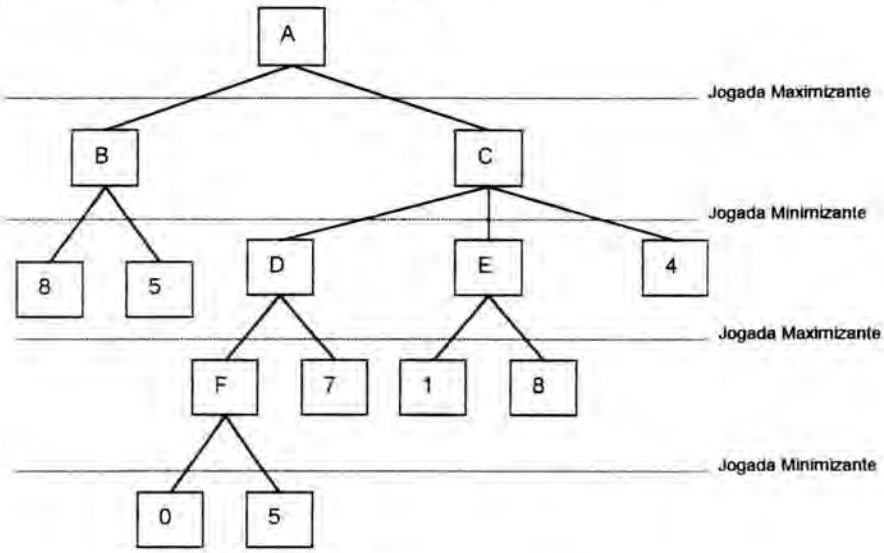
FIGURA 3.1- Tratamento tipico de linha de comandos C

Quando um programador C experiente precisa acrescentar um novo comando a ser processado, possui o conhecimento das convenções e o tratamento habitual deste problema que lhe permite concentrar-se, especificamente, na forma em que cada comando é processado, e não em cada sentença particular do algoritmo. Estudando este comportamento, Adelson demonstrou que programadores novatos, tem a tendência a se concentrar nos aspectos sintáticos de cada linha de código até gradualmente compreender sua função dentro de um algoritmo completo [ADE 81]. Isto implica que a experiência facilita a assimilação de nova informação, baseada em conhecimento preexistente na memória de longo prazo. Soloway e Ehrich [SOL 84] postulam que o conhecimento de programação é constituido por regras gerais independentes da linguagem de programação (regras do discurso) e por planos de programas. As regras do discurso definem propriedades de estilo que um programa deveria satisfazer, como por exemplo nomes de variáveis de acordo com a sua função. Os planos de programas representam, basicamente, uma associação entre objetivos de computação e determinadas estruturas abstratas de programação que satisfazem esses objetivos. Habitualmente, um programa se compõe de diferentes planos intercalados de menor nivel de abstração que satisfazem os diferentes sub-objetivos nos quais o objetivo global é decomposto. Por exemplo, o exemplo acima mostra, em sintaxe C, o plano de percurso seqüencial de um conjunto de itens (*while(set not empty){....}*) combinado com o plano de percurso de uma seqüência de elementos (**++argv*).

O conhecimento de planos de programação constitui uma estrutura de conhecimento que tende a ser comum entre diferentes programadores. Este conhecimento facilita a comunicação e compreensão de programas em um maior nivel de abstração. A violação de regras de discurso ou de planos de programação produz código que é tão dificil de compreender para especialistas quanto para novatos [CUR 89].

As caraterísticas da linguagem de programação, como por exemplo a sintaxe, também contribuem para facilitar ou dificultar a compreensão de um programa. Linguagens dinâmicas, com uma alta potência expressiva, permitem expressar soluções para problemas complexos de forma muito sintética. Esta capacidade é de grande utilidade para o desenvolvimento de programas, mas também torna esses programas mais dificeis de compreender. Linguagens como C, por exemplo, fornecem operadores de manipulação de endereços que podem ser combinados livremente em uma mesma expressão. Isto permite expressar manipulações de estruturas complexas em uma simples linha de código ou a ativação indireta de funções, as quais podem conduzir a comportamentos potencialmente errados.

LISP é um outro exemplo de linguagens dinâmicas com alto poder de expressão. A ausência de tipos de dados, a utilização de convenções de nomes inadequadas e a sintaxe baseada em uma única representação de listas, contribui em muito a uma pobre legibilidade do código. Além disso, a capacidade de avaliação indireta de funções, combinada com soluções tipicamente recursivas impõe uma alta carga cognitiva para compreender a funcionalidade de um dado algoritmo. Por exemplo, as funções seguintes implementam o critério de seleção conhecido em teoria de jogos como *minimax*. Dada uma árvore de jogo, que representa as diferentes alternativas de decisão, o jogador minimizante (alternativamente o maximizante) seleciona o valor que minimiza (maximiza) sua perda (ganho). Este critério implica na seleção alternativa do máximo ou mínimo (mínimo ou máximo) do conjunto de nodos de cada nivel da árvore, dependendo do tipo de jogador.



(A (B (8 5)) (C (D (F (0 5)) 7) (E (1 8)) (4)))

FIGURA 3.2- Representação da árvore do jogo

Com a representação mostrada na FIGURA 3.2 o programa que implementa a busca *minimax* é o seguinte [WIN 83]:

```
(minmax (l)
   (cond
      ((null l) nil)
      (t (map 'min '( maxmin l))) ; retorna o maior dos valores da lista
                                  ; resultante de selecionar os menores das
                                  ; sublistas.
```

```
))

(maxmin (l)
    (cond
        ((null l) nil)
        (t (map 'max '(minmax l)) ; retorna o menor dos valores da lista
                                  ; resultante de selecionar os maiores das
                                  ; sublistas.
))
```

O jogador maximizante avaliará a árvore começando pela função *minmax*, para selecionar o máximo dos mínimos, enquanto o jogador maximizante começará pela função *maxmin*. Como é simples observar, compreender este algoritmo não é trivial. A sua expressão em termos de mapeamentos de segunda ordem que envolvem chamadas recursivas cruzadas, obriga a análise passo a passo do funcionamento do algoritmo (provavelmente utilizando alguma árvore exemplo concreta) até descobrir que se trata, em essência, de uma implementação alternativa do percurso em profundidade de uma árvore. A expressão alternativa do plano habitual do algoritmo, complica em muito a compreensão imediata do seu funcionamento.

3.1.2 A orientação a objetos

Os programas orientados a objetos são, no caso geral, mais difíceis de compreender que os sistemas procedimentais tradicionais (de arquitetura funcional). Neste caso, a estrutura da descrição da solução através da linguagem de programação, corresponde de forma direta com a forma em que esse sistema é executado pelo computador. Assim, representações hierárquicas, como por exemplo, o diagrama de projeto estruturado, oferecem uma representação visual bastante aproximada do modelo mental que um programador possui de um programa em execução. Esta correspondência simplifica a identificação das partes do código que produzem um efeito determinado na execução de um sistema[1]. Uma vez recuperado o modelo funcional, o qual não é necessariamente um tarefa simples, é relativamente fácil raciocinar acerca do funcionamento do sistema e determinar qual a porção de código a ser modificada.

No caso dos programas orientados a objetos, entretanto, a correspondência entre estrutura do código e modelo de execução não é necessariamente direta, sendo necessário tomar em consideração duas perspectivas: a dinâmica e a construtiva.

A visão dinâmica básica dos programas orientados a objetos está constituída por objetos que se comunicam através da troca de mensagens. Os objetos são criados e destruídos dinamicamente durante a execução do sistema. A estrutura típica de um programa orientado a objetos em execução é a de um grafo, com nodos representando objetos, e elos representando referências a objetos. Sendo os objetos criados e destruídos dinamicamente a topologia deste grafo muda dinamicamente. Este modelo é o modelo mental que, habitualmente, um programador utiliza quando constrói um programa orientado a objetos.

Entretanto, os programas orientados a objetos não se constroem através da configuração de objetos e referências a objetos [2], mas através da utilização de conceitos tais como classes, polimorfismo e herança. É através de estes conceitos que se descreve a estrutura em tempo de execução desejada e se satisfazem os objetivos de projeto como extensibilidade e reutilizabilidade. Assim, a perspectiva de construção coloca a ênfase nos elementos utilizados para construir os componentes operacionais, ou seja, classes e sua organização em hierarquias de herança. Uma classe é

---

[1] Esta correpondência é efetivamente válida para programas seqüenciais. Programas paralelos ou concorrentes, agregam a dificuldade de compreender o comportamento de cada processo individual e os mecanismos de sincronização e comunicação que habilitam a interação entre eles. No entanto, cada processo sequencial corresponde com este modelo.

[2] Esta afirmação não é absolutamente correta para o caso de linguagens baseadas em delegação, como por exemplo Self.

um molde para os objetos que são suas *instâncias*. A classe de um objeto descreve os métodos que definem seu comportamento e as variáveis que definem o estado associado. As classes são os elementos de construção básicos que os programadores dispõem para construir suas aplicações. O conceito de herança habilita as classes se organizarem em hierarquias de generalização/especialização, com aquelas classes que são abstratas perto da raiz e as classes concretas como folhas.

Estes mecanismos de construção permitem a fatoração de um sistema em componentes reutilizáveis, os quais se materializam em tempo de execução no modelo mais simples de rede de instâncias. A FIGURA 3.3 ilustra como os caminhos ao longo de uma árvore de herança podem ser resumidos por folhas que definem o comportamento em tempo de execução. Conceitualmente, cada objeto possui sua própria cópia das propriedades (métodos e variáveis) definidas na hierarquia.

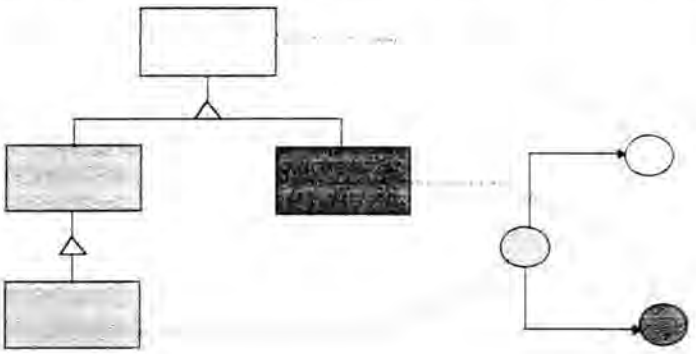


FIGURA 3.3- Correspondência entre as perspectivas construtiva e dinâmica

Deste modo, para compreender uma aplicação orientada a objetos é necessário compreender a visão dinâmica, a construtiva, e os relacionamentos que se estabelecem entre elas. Isto implica na construção do mapeamento inverso desde o modelo simples de rede de instâncias ao modelo de hierarquias de classes que o descreve. Esta dicotomia entre ambos modelos é um dos aspectos que tornam os programas orientados a objetos mais difíceis de compreender que os programas convencionais.

Wilde e Huitt apontam a *distribuição de funcionalidade* como um outro aspecto importante que diferencia os programas orientados a objetos do programas convencionais [WIL 92]. Os programas orientados a objetos tendem a distribuir o código que implementa uma dada função entre um conjunto de métodos pequenos os quais computam uma pequena parte da função total. Um grande número destes métodos apenas transfere uma mensagem para um outro objeto após realizar alguma transformação sobre os argumentos. Assim, os programas tendem a estar formados por um número grande de pequenas unidades de processamento, ao invés do que por um conjunto relativamente pequeno de módulos grandes, como acontece habitualmente com os programas procedimentais. Esta distribuição tem grandes implicâncias para a compreensão detalhada de um programa. Compreender uma simples linha de código pode implicar no seguimento de uma cadeia de métodos através de várias classes diferentes tanto para acima como para abaixo dentro da hierarquia de herança (FIGURA 3.4).

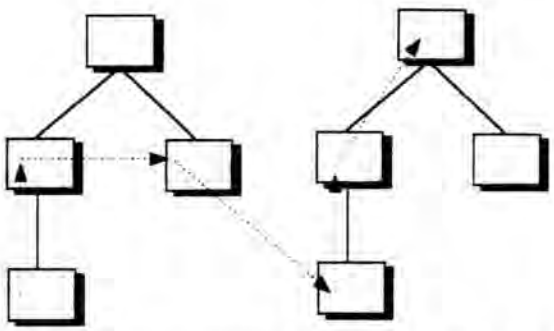


FIGURA 3.4- Seguimento típico da implementação de um método

3.1.2.1 Polimorfismo e acoplamento dinâmico

O polimorfismo e o acoplamento dinâmico de métodos são os mecanismos essenciais que habilitam a construção de software flexível. Também, eles são dois fatores essenciais que contribuem para aumentar a dificuldade de compreensão.

Polimorfismo é a habilidade de uma variável, ou referência a procedimento, para assumir valores de tipos diferentes. O polimorfismo é uma noção que se confunde, freqüentemente, com a sobrecarga (*overloading*) de nomes [CAR 85]. O polimorfismo puro, presente em linguagens do estilo de Smalltalk, implica que um mesmo segmento de código pode tratar objetos de tipos diferentes. Nas linguagens tipadas, como por exemplo *Eiffel*, o polimorfismo adota as formas denominadas de polimorfismo de inclusão e polimorfismo paramétrico. Neste caso um mesmo método pode tratar uniformemente objetos pertencentes a qualquer subclasse da classe na qual está definido. Uma referência polimórfica, por sua vez, é uma variável que pode referenciar objetos de diferentes tipos em diferentes estados da execução. Nos programas orientados a objetos, o método objetivo "real" de uma mensagem dependerá da classe do objeto que está recebendo essa mensagem. A eleição do método a executar pode determinar-se em tempo de compilação com base no tipo (classe) da variável. Se uma referência é polimórfica pode assumir dinamicamente valores de diferentes tipos, então, o polimorfismo se consegue através do acoplamento dinâmico. Sem acoplamento dinâmico, o polimorfismo é uma conveniência sintática equivalente à sobrecarga de nomes presente em linguagens como *Ada* por exemplo.

O polimorfismo, o acoplamento dinâmico e a herança permitem a definição de comportamentos muito complexos dentro da mesma hierarquia de classes, os quais só podem ser compreendidos com exatidão em tempo de execução. Um exemplo deste tipo de comportamentos é representado por classes projetadas para acrescentar funcionalidade dinamicamente a objetos, denominadas *wrappers* [GAM 94]. A maior parte do comportamento de um *wrapper* consiste em propagar para seu componente ou seu recipiente as mensagens que recebe. Isto produz a execução de diferentes redefinições de um mesmo método dentro de diferentes níveis da mesma hierarquia. Qual o método a ser executado depende da configuração de instâncias em cada ponto da execução. Este comportamento, conhecido como o efeito *yo-yo* [TAE 89], torna difícil determinar estaticamente qual o método a ser executado desde uma dada chamada, obrigando a analisar o código de múltiplas classes até compreender o relacionamento entre os diferentes componentes da hierarquia.

Um outro problema relacionado com o polimorfismo é a utilização de nomes inconsistentes dentro de um mesmo sistema. O polimorfismo habilita atribuir o mesmo nome a métodos com comportamentos potencialmente muito diferentes. Assim, um programador pode facilmente interpretar erroneamente a funcionalidade de um dado método presumindo que uma mensagem cumpre uma função sugerida pelo nome de um método. Esta situação é ainda pior na ausência de informação de tipo. Neste caso, não é possível determinar estaticamente qual das múltiplas implementações de uma mensagem enviada é executada, sendo necessário realizar a análise dinâmica da aplicação para identificar a implementação correta.

3.2 Compreensão de Frameworks

Começar a utilizar um framework para construir aplicações pode ser uma tarefa extremamente custosa, ainda que dispondo de documentação apropriada. O problema habitual que os usuários de um framework encontram é que, para conseguir especializar as classes abstratas e descrever como a aplicação se constrói a partir dessas classes, precisam, geralmente, compreender o projeto detalhado das classes do framework. Em frameworks complexos, estas classes codificam padrões de colaboração entre instâncias, através de estruturas de projeto *flexíveis*, isto é, estruturas que permitem *adaptar* o comportamento *geral* provido pelo framework.

Um framework representa um compromisso entre uma solução geral e uma solução flexível. Uma solução geral pode tratar, sem mudanças, diferentes variantes de um dado problema. Uma solução flexível, por sua vez, é uma solução que através de pequenas mudanças na estrutura pode ser

adaptada para resolver essas diferentes variantes. Soluções gerais são certamente desejáveis, mas na maioria dos casos, apresentam problemas de desempenho ou são limitadas a domínios muito restritos [PAR 79]. Soluções flexíveis podem ser adaptadas para cada caso particular, permitindo explorar ao máximo aqueles aspectos que simplificam sua solução em termos de performance e funcionalidade. Mas, no caso geral, estruturas de projeto muito flexíveis implicam em projetos altamente complexos e, portanto, difíceis de compreender.

Para ilustrar com maior clareza os diferentes aspectos que tornam complicada a compreensão de frameworks, apresenta-se, a seguir, um exemplo real de uso do framework para construção de interfaces visuais, MVC, provido pelo ambiente *Smalltalk-VisualWorks* [PAR 94].

3.2.1 Um cenário real: Gerenciamento de visões em Smalltalk

O framework MVC representa um dos exemplos mais relevantes, existentes na atualidade, de estrutura de projeto adaptável dinamicamente, e de fatorização de abstrações que maximizam a reutilizabilidade de funcionalidade por composição de objetos. O grau de padronização de interfaces e encapsulamento de funcionalidade permite a construção de interfaces de diálogos com um editor de composição visual, através do qual um programador pode descrever a composição da interface sem necessidade de programação.

Para o caso de sistemas de visualização de software, por exemplo, o suporte provido pelo framework permite construir visualizações bastante complexas com relativo pouco esforço. Mas, neste caso, é necessário compreender a organização do framework para construir a visualização desejada.

3.2.1.1 Adaptação do mecanismo de redimensionamento

A generalidade dos mecanismos de redimensionamento e deslocamento de componentes visuais geram problemas de eficiência para construir visualizações muito variáveis, as quais precisam ser atualizadas automaticamente em tempo real. A FIGURA 3.5 apresenta um exemplo deste tipo. Esta visualização representa uma classe de um framework que agrupa os métodos de acordo com sua categoria (abstratos, *hook*, *template*, base). A parte inferior da figura mostra a estrutura de instâncias aproximada que gerarão a visualização em tempo de execução. Cada tipo de componente visual é

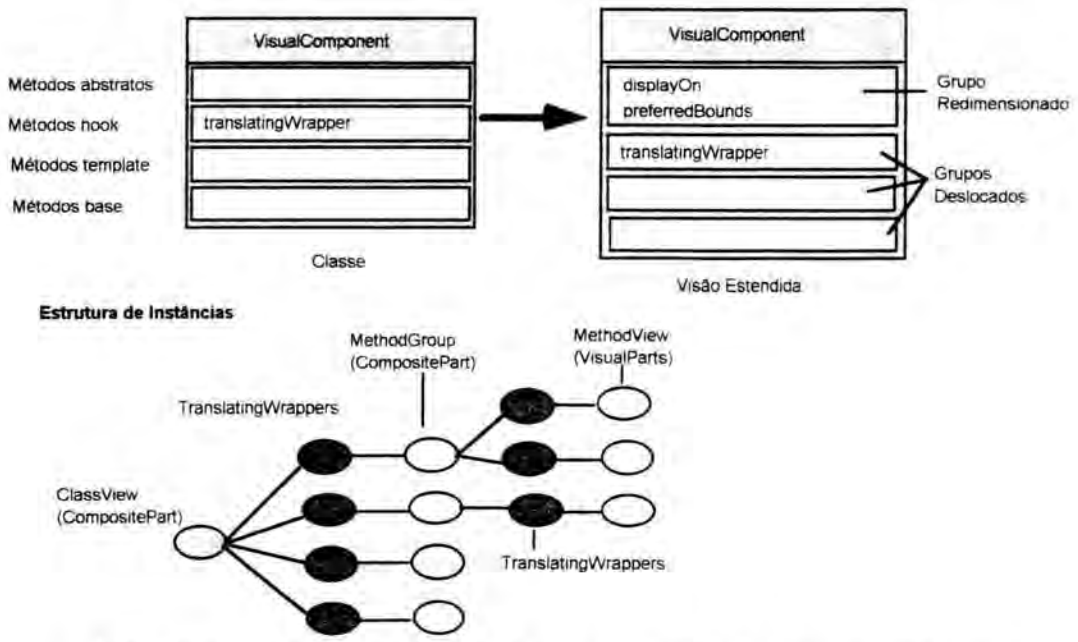


FIGURA 3.5 - Uma visualização de classe e estrutura de instâncias que a gera

construído como subclasse de alguma das classes do framework mostradas entre parênteses. Esta visualização é gerada automaticamente por uma ferramenta que coleta informação de um programa em execução (ver capítulo 7). Quando um método é ativado pela primeira vez, seu nome é acrescentado ao grupo que contém os métodos de sua categoria. Isto requer o redimensionamento do grupo em questão e o deslocamento dos grupos restantes para manter a distribuição espacial sem sobreposição.

Uma vez que um novo método é acrescentado, digamos *displayOn:*, é necessário deslocar os outros grupos para manter a organização espacial. Para isto, *CompositePart* fornece o método: *move: aWrapper to: aPoint* que produz o deslocamento da componente visual representado pelo argumento *aWrapper* ao ponto representado pelo argumento *aPoint*, mas o mecanismo disparado pela utilização deste método torna a geração da visualização intolerável em termos de desempenho.

Não possuindo um conhecimento detalhado do framework, o programador enfrenta a necessidade de determinar a causa desta ineficiência através da análise da estrutura do código que implementa esta funcionalidade. A seguir apresenta-se a seqüência resumida dos passos que o desenvolvedor deve realizar para determinar uma solução para este problema. O código é apresentado ressaltando as chamadas a métodos importantes em negrito e itálico, resumindo alguns passos para simplificar a apresentação do exemplo.

A seqüência começa quando a visualização da classe, representada pela classe *ClassView*, recebe a mensagem *move:to:*, herdada de *CompositePart*.

ClassView(CompositePart):

move: aWrapper to: aPoint
"Move the bounds of aWrapper to aPoint."
| old |
(self components includes: aWrapper)
ifFalse: [^self error: 'invalid component'].
old := aWrapper bounds.
self newBounds: (aPoint extent: old extent) oldBounds: old forWrapper: aWrapper.

O método *newbouds*.. informa ao componente visual a ser deslocado sua nova posição para depois calcular sua nova área em função do deslocamento de um dos seus componentes (*computePreferredBounds*)

newBounds: newBounds oldBounds: oldBounds forWrapper: aWrapper
| lastPreferredBounds |
aWrapper newBounds: newBounds containingBounds: self bounds.
lastPreferredBounds := self preferredBounds.
self computePreferredBounds.
lastPreferredBounds = preferredBounds
ifFalse: [self changedPreferredBounds: lastPreferredBounds].
.....

O wrapper do componente deslocado calcula o novo ponto de origem solicitando ao seu componente os limites (bounds) atuais, para depois informá-lhe os novos limites.

TranslatingWrapper(1):—

newBounds: newBounds containingBounds: containingBounds
"The receiver's new bounds is to be newBounds which is a rectangle relative to containingBounds. Update the receivers layout."

| delta |
delta := newBounds origin - self ***bounds*** *origin.*
self setOrigin: self translation + delta rounded.
self ***setComponentBoundsTo****: newBounds.*

O grupo de métodos solicita ao seu recipiente os limites que tem atribuídos, através da mensagem *compositionBounds*.

MethodGroup(VisualPart)(1):

```
bounds (*)
    "Answer the receiver's compositionBounds if there is a container, otherwise
        answer preferredBounds."
    ^container == nil
        ifTrue: [self preferredBounds]
        ifFalse: [container compositionBoundsFor: self]
```

O controle retorna ao wrapper, que propaga a mensagem para acima na hierarquia de visões e retorna os limites transladados.

TranslatingWrapper(2):

```
compositionBoundsFor: aVisualPart
    "The receiver is a container for aVisualPart An actual bounding
        rectangle is being searched for aVisualPart. Forward to
        the receiver's container."
    ^(container compositionBoundsFor: self)
        translatedBy: self translation negated!
```

O recipiente do grupo de métodos é a visão da classe, a qual herda a implementação do método *compositionBoundsFor:* de CompositePart. Este método retorna o valor dos limites definidos pelo componente a ser deslocado.

ClassView(CompositePart)(2):

```
compositionBoundsFor: aVisualComponent
    "No Wrappers between the receiver and the
    originator implement a compositionBounds.
    Answer the perferredBounds."

    ^aVisualComponent preferredBounds.
```

O controle é passado novamente ao grupo de métodos a ser deslocado, o qual retorna os seus limites atuais (os quais supõe-se foram calculados previamente).

MethodGroup(CompositePart)(2):

```
preferredBounds
    "Answer the merged bounds of the receiver's
        components"

    ^preferredBounds == nil
        ifTrue: [self computePreferredBounds]
        ifFalse: [preferredBounds]! !
```

.....

O controle retorna ao wrapper quem informa ao grupo de métodos os novos limites calculados através da mensagem *bounds:*.

TranslatingWrapper(1):

```
setComponentBoundsTo: newBounds
    component bounds: (newBounds translatedBy: self translation negated)
```

A implementação herdada de CompositePart do método bounds: calcula a nova dimensão do grupo de métodos reorganizando a estrutura interna através da mensagem *layoutComponentsForBounds*. Esta reorganização implica no cômputo da área resultante da suma das áreas ocupadas pelas visões dos métodos.

MethodGroup(CompositePart) (1) :

```
bounds: newBounds
```

```
        super bounds: newBounds.
        self layoutComponentsForBounds: (0@0 extent: newBounds extent)!

    layoutComponentsForBounds: newBounds
        "The receiver has been sized to the given parameters. Re-layout all of the receiver's
            components."
        preferredBounds := 0@0 extent: 0@0.
        1 to: components size do:
            [:i |
            | box component |
            component := components at: i.
            box := self
                computeDisplayBoxFor: component
                inDisplayBox: newBounds.
            component bounds: box.
            preferredBounds := preferredBounds merge: box].

    computeDisplayBoxFor: aComponent inDisplayBox: aRectangle
        "Answer a rectangle for aComponent to use as its display box."

        ^aComponent rectangleRelativeTo: aRectangle

MethodView(VisualComponent)(1):
        rectangleRelativeTo: containingBoundingBox
            "Answer a rectangle for the receiver relative to the
                containingBoundingBox."

            ^self bounds translatedBy: containingBoundingBox origin.
```

Neste ponto da execução o método bounds é chamado, repetindo-se a seqüência marcada com (*) acima, a qual terminará solicitando os limites desejados à visão do método. O custo computacional do processo de cálculo desta região depende linearmente do número de métodos existentes em cada grupo. Assim, se considera-se, na media, que cada grupo terá um numero *m* de métodos e assumindo o custo de envio de uma mensagem sendo de valor 1, o custo aproximado para deslocar *n* grupos[3]:

$$\sum_{0}^{n} m*(m-1) \equiv (n-1)*m*(m*(m-1)) = (n^2 - n)*(m^2 - m) = n^2*m^2 - n^2*m - n*m^2 + n*m.$$

Assumindo $m >> n$ **o custo aproximado será**

$$n^2*m^2 - n*m^2$$

Se considera-se que o custo total de uma visualização complexa será proporcional ao número de classes presentes na visualização, este cálculo torna a geração da visualização gradualmente mais ineficiente.
Finalmente a visualização da classe calcula seu tamanho final através da mensagem computePreferredBounds, a qual executa novamente a seqüência marcada com (*).

```
    computePreferredBounds
        "Compute the receiver's preferredBounds"

        preferredBounds := 0@0 extent: 0@0.
        1 to: components size do:
            [:i |
            preferredBounds := preferredBounds merge: (components at: i) bounds].
        ^preferredBounds
```

[3] Este cáculo de custo é muito aproximado, servindo só aos fins de motivar a problemática do exemplo.

O ponto relevante deste exemplo é que, após toda a seqüência de passagem de mensagens mostrada, o resultado final é só a atualização do ponto de origem do wrapper associado com o grupo de métodos deslocado. Não tendo sido modificado o conteúdo deste grupo, o cálculo da região ocupada é absolutamente desnecessário e pode ser evitado simplesmente redefinindo a mensagem *bounds:* na classe MethodGroup, para fixar sua nova área, na variável de instância preferredBounds (definida na superclasse *CompositePart*). O método *bounds:* é, na realidade, um método *hook* definido na classe abstrata *VisualComponent* com uma implementação vazia, isto é, o seu comportamento *default* é não produzir efeito nenhum:

```
bounds: newBounds
    "An actual bounding rectangle is being asserted.
    Many VisualComponents do nothing."
    ^self
```

A redefinição deste método permite evitar o processo desnecessário a um custo computacional constante, o que reduz o tempo necessário para gerar uma visualização em um fator de 10 a 1 aproximadamente. Entretanto, para realizar esta redefinição, o programador precisa identificar a fonte do problema e determinar a solução em base a análise detalhada do mecanismo de deslocamento.

Este exemplo mostra vários aspectos relevantes a respeito da problemática da compreensão de frameworks. A classe abstrata *VisualComponent*, define dois métodos abstratos os quais devem ser implementados por subclasses concretas. Mas, a simples implementação destes métodos, não garante os resultados desejados para uma implementação particular. Para resolver este problema o programador necessita compreender a organização dinâmica de visões, para poder assim realizar o mapeamento às múltiplas classes nas quais se distribui a definição do comportamento dessas instâncias. Este comportamento é redefinido em diferentes subclasses dentro da mesma hierarquia, as quais provem diferentes implementações de métodos *hook*. Deste modo, não é suficiente só compreender a interface definida pelas classes, mas também é necessário compreender a hierarquia de componentes concretos nos quais cada abstração é especializada.

Compreender estes mecanismos através da análise do código é uma tarefa custosa e demorada, especialmente sem ferramentas que facilitem a tarefa. Particularmente, a seqüência de controle mostrada não está documentada nos manuais de utilização, já que envolve métodos privados de componentes prontos para serem reutilizados. Assim, o código fonte é a única fonte de referência. No caso de Smalltalk, o mecanismo habitualmente utilizado por programadores é determinar aproximadamente algum ponto no código onde começar a análise, e gerar uma exceção que ativa o depurador interativo para realizar o seguimento passo a passo da execução, podendo inspecionar em qualquer momento o código sendo executado e as instâncias envolvidas. Este mecanismo, no entanto, é rudimentar e não adequado para problemas muito complexos.

3.3 Técnicas de Modelagem e Documentação

Um outro problema relacionado com a compreensão de frameworks é a dificuldade que os desenvolvedores encontram para documentá-los [LAN 95]. A documentação de frameworks, apresenta problemas de maior complexidade que a documentação de uma sistema convencional, seja este orientado ou não a objetos. Esta complexidade pode ser atribuída, essencialmente, a duas razões: a natureza genérica ou abstrata do projeto e o objetivo de qualquer framework que é a sua reutilização. Por um lado, a documentação deve mostrar como utilizar o framework para construir aplicações, sem entrar em detalhes de como é o funcionamento interno dele. O principal interesse de um usuário é saber o que fazer para utilizá-lo, e não como foi projetado. Por isto, a documentação deve explicar, através de exemplos, a forma em que um usuário pode resolver diferentes problemas usando as funções do framework, quais classes devem ser especializadas, que métodos devem ser implementados e quais mensagens devem ser enviadas a cada classe envolvida.

Por outro lado, a descrição do projeto detalhado do framework também é necessária. Enquanto a documentação de uso serve aos fins dos usuários ocasionais, a documentação do projeto

ajuda para utilizar o framework para aplicações além das previstas pelo projetista original [JOH 92]. A compreensão clara da forma em que um framework trabalha é de grande ajuda para utilizá-lo em todo o seu potencial.

Um enfoque adotado pela maioria dos modelos de Análise e Projeto (ADOO), é estender os diagramas que representam relacionamentos de herança entre classes, com conceitos provenientes do modelo de Entidade-Relacinamento. Como exemplos podem citar-se OMT [RUM 91] e OOA [COA 90]. Os modelos baseados no modelo ER, provêm um paradigma consistente para modelar, em um alto nivel de abstração, a estrutura genérica da topologia das instâncias em tempo de execução, sendo de utilidade para compreender como as diferentes classes de um sistema relacionam-se umas com as outras.

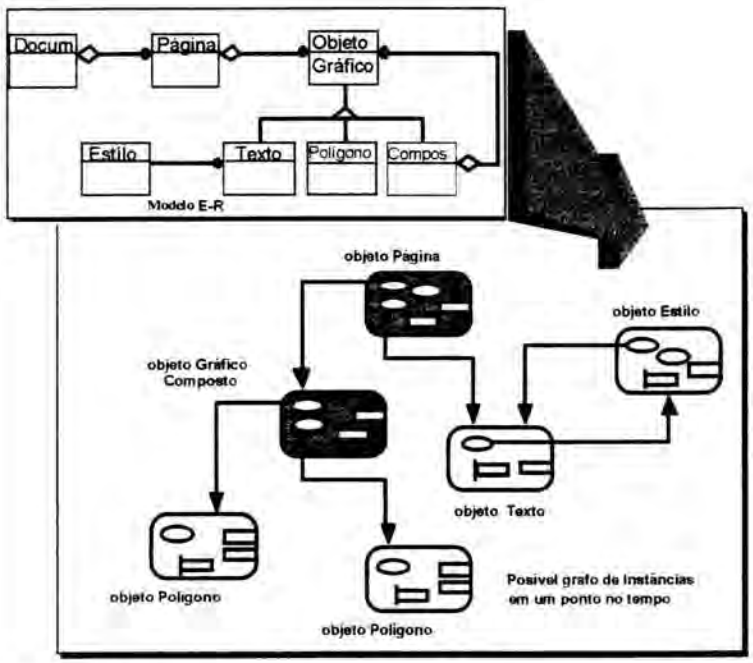


FIGURA 3.6- Correspondência entre o modelo ER e topologia de instâncias

O modelo ER, porém, é estático e carece da potência descritiva necessária para modelar aspectos importantes do comportamento do sistema, como por exemplo, padrões de fluxo global e local, e grupos colaborativos. Por esta razão, vários métodos completam o modelo estático provido pelo ER, com uma visão complementar baseada em máquinas de transição de estados para modelar o comportamento interno das instâncias [RUM 91][SHL 88][COL 92]. Cada objeto, instância de alguma das classes descritas no diagrama ER, é modelada como uma máquina de transição de estados. Estas máquinas representam as transições de estados produzidas pelo envio de mensagens entre objetos. Os objetos realizam ações durante as transições de estados ou enquanto permanecem nesse estado.

Esta abordagem, embora útil para descrever o comportamento de objetos particulares, apresenta limitações para expressar o comportamento conjunto de grupos de objetos, ocultando as interações entre eles nas ações realizadas durante as transições de estados. Isto representa uma forte limitação pois é muito difícil deduzir o comportamento do sistema completo a partir do comportamento de objetos individuais [COL 92]. Por outro lado, o enfoque: *cada objeto possui uma máquina de estados*, pode levar a uma especificação de comportamento fragmentada e granularidade muito fina. Estas limitações, dificultam a descrição do comportamento agregado de um grupo de objetos, e a compreensão dos padrões de comportamento codificados por um framework.

Os diagramas de seqüenciamento de mensagens, como os diagramas de interação de Jacobson e os diagramas de tempo de Booch (FIGURA 3.7), são técnicas mais adequadas para

descrever o comportamento de grupos de objetos projetados para trabalhar juntos. Sintaticamente, os diagramas de interação não representam o retorno explícito do controle, mas ele está implícito na seqüência de mensagens. Portanto, os diagramas de interação, como os diagramas de tempo, pressupõem que a responsabilidade para o fluxo de controle tem sido atribuída a clientes e servidores.

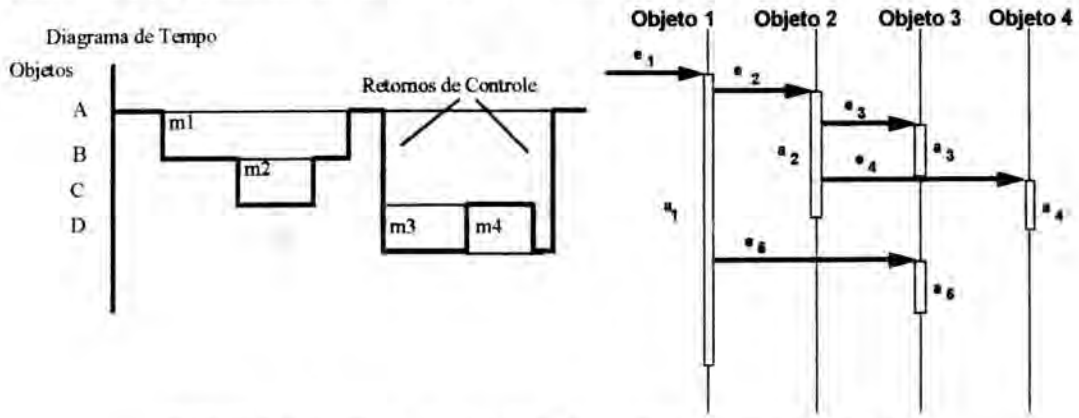


FIGURA 3.7 - Representações de fluxo de controle entre objetos

Este tipo de representação é mais adequado para descrever mecanismos de colaboração complexos, mas apresenta uma grande limitação para descrever o fluxo abstrato codificado por classes abstratas, o qual é especializado dentro da hierarquia de subclasses. Este fluxo baseia-se em mensagens ao próprio objeto, os quais sobrepõem-se no tempo. Este mecanismo não pode ser claramente representado por este tipo de diagramas. A mesma limitação acontece entre objetos que não interagem estritamente segundo o modelo cliente-servidor, como é o caso dos *wrappers* e os componentes visuais do exemplo anterior. Neste caso o controle flui várias vezes entre os mesmos componentes da interface, os quais se enviam a si mesmos como parâmetros de mensagens polimórficas. Estas interações são difíceis de modelar utilizando este tipo de representações temporais, na forma em que são definidas, obrigando a realizar modificações para expressar estes casos particulares.

A importância de notações gráficas é amplamente reconhecida para descrever a estrutura e funcionalidade de um sistema de software. Cada método de desenvolvimento orientado a objetos define suas próprias notações para descrever a estrutura estática e dinâmica dos sistemas, mas não suportam *adequadamente* a descrição de frameworks orientados a objetos. Parte desta dificuldade pode ser atribuída ao fato de que as notações foram desenvolvidas com objetivo de descrever sistemas orientados a objetos específicos e não para sistemas que vão ser reutilizados como um todo. Como conseqüência disto, não fornecem mecanismos adequados para ressaltar aspectos que são próprios dos frameworks, como por exemplo, quais partes devem ser redefinidas, qual o comportamento genérico, quais os comportamentos abstrato e o base, e como se relacionam as classes através do fluxo de controle genérico.

Os frameworks existentes na literatura geralmente são descritos utilizando variações de alguma das notações conhecidas, como OMT [RUM 91] ou grafo de colaborações [WIR 90], por exemplo. Esta descrição é complementada com narrativa ou, excepcionalmente, com alguma técnica especial como contratos ou padrões textuais. Em alguns casos, as notações originais são modificadas para ressaltar tipos de classes e de métodos. Por exemplo, algumas modificações a OMT utilizam itálicas para diferenciar os métodos abstratos dos concretos, e as classes abstratas das concretas. Estas variantes, ajudam a compreender melhor as caraterísticas das classes, mas a informação acrescentada não é suficiente para compreender diretamente a estrutura genérica fundamental de um framework.

3.3.1 Técnicas específicas para documentação de frameworks

As limitações das técnicas de descrição provida pelas metodologias mais difundidas, levou vários autores a proporem diferentes técnicas especialmente projetadas para documentar frameworks. Estas técnicas mudam o foco de atenção para as interações que se produzem entre grupos

∈: pertence	∉: não pertence	=: atribuição
⇒: implica	→: mensagem	∀: para todo
||: operador paralelo ; : operador seqüencial		
<*operador variável* : *predicado ações*> : aplicação das *ações* a todos os elementos do conjunto descrito pelo *predicado*, unidas pelo *operador*, ou seja, aplicação paralela ou seqüencial.		
{*ações* } : pre e post condições.		

FIGURA 3.8 Operadores especiais da notação de contratos

de objetos, ao invés de se centrar no comportamento de objetos individuais. Nesta seção descrevem-se, sinteticamente, as abordagens mais relevantes propostas na literatura: contratos [HEL 90], sistemas de padrões textuais [JON 92], padrões de projeto [GAM 94] e meta-padrões [PRE 94]. A descrição é realizada sem entrar em considerações a respeito das suas vantagens e limitações, as quais são analisadas em conjunto no final do capítulo.

3.3.1.1 Contratos

A técnica de contratos (*contracts*) [HEL 90] é uma técnica textual baseada em uma linguagem semi-formal para descrever os relacionamentos de colaboração entre classes de um framework. Um contrato descreve as colaborações que existem entre um conjunto de participantes, através de uma linguagem semi-formal. Com esta linguagem especificam-se os participantes na relação e suas obrigações contratuais como variáveis que devem possuir, a interface externa que devem prover e a seqüência ordenada de ações que devem instanciar certas condições como verdadeiras em resposta às mensagens. Opcionalmente, pode-se especificar um invariante que os participantes no contrato colaboram para manter. O exemplo seguinte mostra a estrutura parcial de uma especificação de um contrato que descreve as colaborações entre o modelo (*Model*) e as visões (*Views*). O contrato estabelece os serviços que tem que suportar cada um dos participantes, os quais deverão ser métodos implementados pelas classes que participarão no contrato.

A notação utilizada consiste das construções especiais mostradas na FIGURA 3.8.

```
CONTRACT ModelView
 Model SUPPORTS [
    // conjunto de atibutos
    atributes []
        // atribuição a variável
    setVarValue(varName,val) ⇒atributes[varName].value;
                // pós-condição: a variável fica com o valor val
                        {atributes[varName].value=val};
                // invoca ao serviço changed para informar a mudança
                        changed()
        // Seqüencialmente cada View recebe a mensagem refresh
    changed() ⇒ <; || v: v Views: v → refresh() >
    addView(v) ⇒ { v Views }
    removeView(v) ⇒{ v Ï Views }
    ]

Views : SET (View) WHERE EACH View SUPPORTS [
        // atualiza a apresentação na tela depois de atualizar os valores
    refresh() ⇒{ View reflects Model.atributes};
                display()
    setModel(m) ⇒{ model=m }
    ]
INVARIANT
        // Cada visão sempre mostra o estado atual do seu modelo
    Model.setVarValue(varName,val)
        < ∀ v: v ∈ Views: v reflects Model.atributes>
INSTANTATION
```

Model→addView(View)
View→ setModel(Model)
END CONTRACT // ModelView

O contrato define dois participantes *Model* e *Views*. *Views* é um conjunto de componentes do tipo *View*, cada um dos quais suporta os serviços *setModel* (que indica para um *View* qual é o seu modelo) e *refresh* que tem como pre-condição que a visão tem que refletir o estado do modelo para executar a operação *display*. O modelo, por sua vez, vai indicar para cada visão associada quando mudou o seu estado, através da mensagem *changed*.

3.3.1.2 Sistemas de padrões

Johnson propõe documentar frameworks utilizando uma notação textual informal, que estrutura a descrição como um conjunto de padrões (***patterns***) organizados hierarquicamente [JOH 92]. Um padrão descreve um problema que se repete no dominio do framework e então descreve como resolvé-lo, seguindo a estrutura seguinte:

- Uma descrição do problema (em itálico)
- Discussão detalhada das diferentes formas possiveis para resolver o problema, com exemplos, diagramas e ponteiros para outras partes do framework, que sejam de utilidade para explicar como o framework resolve o problema.
- Resumo da solução (também em itálico) seguido de ponteiros para outros padrões que completam a descrição

O exemplo a seguir apresenta um padrão que descreve o processo de interpretação de eventos de mouse em um framework de interfaces:

Pattern 9: Interpretando eventos

Nas interfaces de manipulação direta, a interação realiza-se apontando com o cursor à apresentação do objeto de interesse e apertando ou libertando um dos botões do mouse. Cada uma dessas ações representa um evento. Deve-se poder especificar que eventos serão reconhecidos sobre cada visão e que ações executar-se-ão nesse caso.

De acordo com o contexto, isto é a apresentação que esteja abaixo do cursor na hora de se produzir o evento, a interpretação deste evento varia. Por exemplo, um *click* do botão esquerdo sobre um item de um menu significa selecionar a opção, sobre um objeto de edição significa selecioná-lo para uma ação posterior, sobre a área de trabalho inserir uma instância do tipo previamente selecionado, etc.

Considerando a estrutura hierárquica de visões, existem apresentações que encontram-se na frente das outras, sendo necessário que as apresentações exteriores interceptem o evento primeiro, antes de que seja tratado pelas que estão atrás. Uma possível forma de obter este efeito consiste em propagar o evento, desde o topo da hierarquia até as subvisões, até que alguma delas tratem-no; se isto não ocorre, então tentar tratar o evento ao retorno. Este é o comportamento implementado na classe *View*.

Para deixar independente as apresentações do manejo dos eventos, encapsula-se seu tratamento em objetos separados encarregados de processar os eventos, e que associam-se dinamicamente a cada apresentador. Estes objetos são os *recognizers* (reconhecedores). Cada *reconhecedor* tem associado um *view* e um *command*. Ao receber a mensagem de que produziu-se um determinado evento, um reconhecedor realizará eventualmente várias ações, por exemplo, enviar uma mensagem ao comando que tem associado, para ativá-lo ou para setar algum input. Um reconhecedor possui dois estados: ativo e suspenso. Para reconhecer um evento um reconhecedor deve estar ativo. Um comando, cujo reconhecedor esteja suspendido, não pode ser ativado.

Para instanciar reconhecedores, deve-se enviar a mensagem createRecognizer à classe de reconhecedor correspondente (ClickRecognizer, DragRecognizer, etc.) com o

tipo de evento que se quer tratar, levando como argumentos o modelo associado ao reconhecedor, a visão, o comando, a mensagem que deve enviar o reconhecedor, o destino e argumentos desta mensagem

A documentação de um framework estará constituída por um conjunto destes padrões, os quais podem ser facilmente compreendidos por usuários não especialistas no framework. Um padrão explica, em uma linguagem simples, o que um usuário deve fazer para resolver um problema utilizando o framework, indicando, por exemplo, qual classe deve ser especializada e que mensagem deve definir para conseguir um efeito desejado.

3.3.1.3 Padrões de projeto

Os padrões de projeto (*design patterns*) são padrões de organização de hierarquias de classes, protocolos e distribuição de responsabilidades entre classes, que caraterizam *construções elementares* de projeto orientado a objetos [GAM 94]. Um *padrão de projeto* é uma estrutura que aparece repetidamente nos projetos orientados a objetos, a qual é utilizada para resolver um problema determinado de forma flexível e adaptável dinamicamente. Por exemplo, a forma em que uma aplicação pode ser independente do tipo de interface gráfica na qual tem que rodar ou como separar a representação dos dados da sua apresentação na tela.

Os padrões de projeto representam um avanço importante na área de orientação a objetos, pois oferecem um catálogo de *planos de projeto* que permitem a reutilização de *soluções* de projeto que provaram ser efetivas para resolver problemas semelhantes. Eles dão um nome e uma descrição abstrata para estas estruturas, permitindo assim comunicar um projeto em termos de uma linguagem de mais alto nível que as notações básicas. A sua descrição num nível abstrato permite sua reutilização para projetar novas aplicações ou melhorar aplicações existentes, de forma independente da implementação. Também, este tipo de informação pode ser de muita utilidade para documentar o projeto de um framework em um alto nível de abstração.

Um exemplo conhecido é o denominado padrão denominado *Composite*. Este padrão descreve a forma de compor objetos em estruturas de árvore para representar hierarquias de partes ou de conteúdo. O padrão define a forma de tratar múltiplos objetos compostos recursivamente como um objeto simples, simplificando assim o código dos clientes. No exemplo do MVC a hierarquia definida por VisualPart representa uma implementação deste padrão, o qual diferencia explicitamente os objetos compostos (*CompositePart*) das folhas (*MethodView*), sendo uma superclasse de ambos (*VisualPart*) a que representa o objeto composto. A FIGURA 3.9 apresenta a estrutura de classes típica deste padrão, para o caso do exemplo. Tipicamente o objeto composto distribui a aplicação de operações entre seus filhos, obtendo uniformidade através do polimorfismo. Este é o caso do método *displayOn:*, cuja implementação no objeto composto consiste de propagar a mensagem aos componentes visuais contidos pela instância de *CompositePart*.

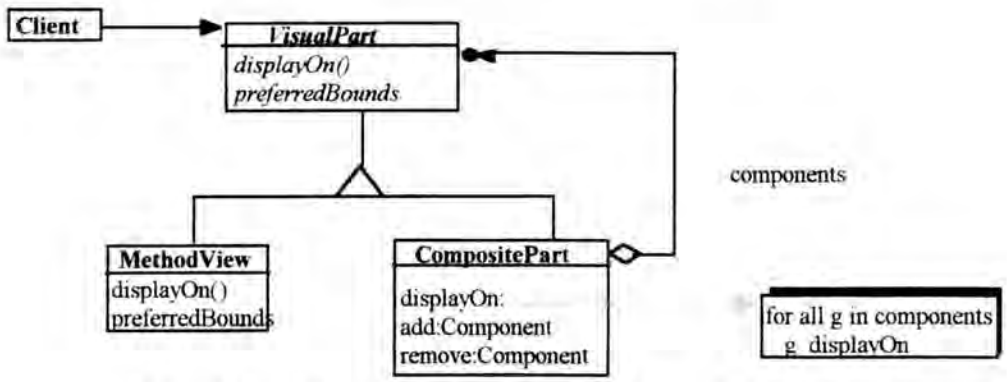


FIGURA 3.9. Estrutura abstrata de classes do padrão Composite

Os padrões de projeto identificados até agora variam na sua granulosidade e nível de abstração, mas mesmo assim, possuem caraterísticas comuns e em alguns casos estão fortemente relacionados. Do ponto de vista da sua utilização é necessário algum esquema que permita classificá-los sistematicamente. [GAM 93] apresenta um sistema de classificação que agrupa os diferentes padrões em categorias que facilitam sua compreensão e utilização. Este sistema divide o conjunto de padrões em função de dois critérios ortogonais: escopo e caracterização. A TABELA 3.1 mostra o espaço de padrões de projeto atualmente identificado, particionado por estes critérios. Cada padrão é descrito informalmente através de uma ficha que lista todas as caraterísticas relevantes para sua utilização.

•**Escopo** é o domínio sobre o qual o padrão pode ser aplicado. Os padrões categorizados dentro da jurisdição de classes tratam acerca de relacionamentos entre classes e as suas subclasses. A jurisdição de classes cobre a semântica estática dos relacionamentos. A jurisdição de objetos envolve relacionamentos entre objetos individuais, enquanto a jurisdição de compostos abrange as estruturas de objetos recursivas. Alguns padrões capturam conceitos que abrangem várias jurisdições, razão pela qual existem versões para cada jurisdição que o padrão abrange. Na tabela pode se observar o caso dos padrões *Adapter* e *Bridge*, os quais aplicam-se tanto na jurisdição de classes como na de objetos, ou *Iterator* que abrange as jurisdições de objetos e compostos.

•**Caracterização** identifica a função do padrão. Os padrões *Criacionais* envolvem o processo de criação de objetos, os *Estruturais* tratam acerca da forma em que classes e objetos são compostos para formar estruturas de nível superior. Os padrões *Comportamentais*, por sua vez, caraterizam a forma em que classes e objetos interagem e distribuem responsabilidades.

TABELA 3.1- Espaço de Padrões de Projeto

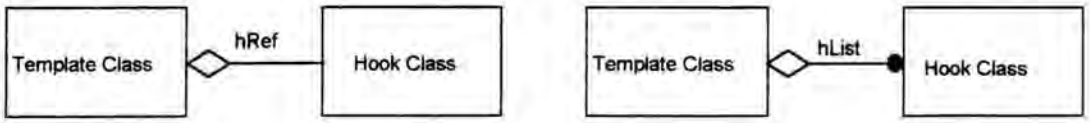
		Caracterização		
		Criacional	**Estrutural**	**Comportamental**
Escopo	**Classe**	*Factory Method*	*Adapter*	*Template Method*
	Objeto	*Abstract Factory* *Prototype* *Singleton*	*Adapter* *Bridge* *Facade* *Proxy* *Flyweight*	*Chain of Responsability* *Command* *Iterator* *Mediator* *Memento* Observer *State* *Strategy* *Visitor*
	Compostos	*Builder*	*Composite* *Decorator*	*Iterartor* *Visitor*

O apêndice 2 apresenta uma descrição sintética de todos os padrões descritos no catálogo.

3.3.1.4 Meta-Padrões

Em um nível mais baixo de abstração que os padrões de projeto, Pree propõe a idéia de meta-padrões como um complemento para documentar a estrutura essencial de um framework [PRE 94]. Baseando-se na premissa que a estrutura essencial de um framework é composta pela organização dos métodos genéricos (*template*) e os métodos redefiníveis (*hook*), Pree identifica um conjunto básico de combinações destes dois conceitos na estrutura de um framework, os quais determinam os meta-padrões. Sob esta premissa, uma classe de um framework é considerada uma classe template se possui um método template, classe hook se possui métodos hook ou classe *template-hook* se possui ambos tipos de métodos. Com estas estruturas básicas, é possível documentar a estrutura essencial de um framework de várias formas e em diferentes níveis de abstração. Uma forma pode consistir, simplesmente, da estrutura de classes *template* e *hook* de acordo com os meta-padrões.

Um método template é aquele que invoca um outro método que pode ser redefinível, ou seja, um método *hook*. Cabe aqui notar, que não se faz diferença entre métodos abstratos e *hook*. Assim, se o método *hook* pertence a uma outra classe pertencente a outra hierarquia, isto determina uma organização denominada Padrão de Conexão (FIGURA 3.10). De acordo com a funcionalidade do relacionamento, este padrão pode ser 1:1 ou 1:N.



Padrão de Coneção 1:1

Padrão de Coneção 1:N

```
T::TH(){
    HookObject-> HookMethod(..);
}
```

```
T::TH(){
    forall HookObject in hList
        HookObject-> HookMethod(..);
}
```

FIGURA 3.10- Padrões de Conexão

Uma forma diferente de relacionamento existe quando os métodos hook pertencem a uma superclasse na hierarquia. Isto conduz a uma conexão recursiva, na qual uma subclasse invoca tipicamente o mesmo método definido na superclasse. A FIGURA 3.11 mostra a estrutura de classes dos padrões denominados Padrões de Conexão Recursiva.

Padrão de Coneção Recursiva 1:1

Padrão de Coneção Recursiva 1:N

```
T::TH()
    HookObject-> TH(..);
}
```

```
T::TH()
    forall HookObject in hList
        HookObject-> TH(..);
}
```

FIGURA 3.11- Padrões de Conexão Recursiva

Finalmente, quando os métodos template e hook, se encontram na mesma classe, os padrões anteriores se combinam nos Padrões de Unificação Recursiva, como mostra a FIGURA 3.12.

FIGURA 3.12- Padrões de Unificação

3.3.2 Análise

As técnicas com formato textual são potencialmente poderosas, mas no caso geral não ajudam a compreender rapidamente a essência de um projeto na mesma forma que os enfoques baseados em gráficos, apesar da natureza semi-formal destes últimos. Especificações formais, por sua vez, podem levar a níveis de detalhe que fazem com que a essência da arquitetura do framework seja difícil de compreender. Dentro das técnicas textuais, os *sistemas de padrões* propostos por Johnson adequam-se melhor para descrever, de um modo simples, a maneira de utilizar um framework, mas tendem a descrever o projeto de um framework implicitamente, impondo o risco de prover informação insuficiente quando surge a necessidade de estender a funcionalidade. Os *contratos* descrevem de forma mais compacta o projeto detalhado do framework, centrando a sua atenção nas colaborações abstratas entre participantes, independentemente das classes que implementarão o comportamento. Deste modo, a estrutura do framework fica descrita em uma linguagem abstrata, que permite a análise do framework em forma independente da linguagem de implementação. Esta abordagem, entretanto, não é de utilidade para explicar como o framework pode ser utilizado para construir aplicações.

Os meta-padrões oferecem um forma sintética para documentar relacionamentos essenciais de classes de um framework, mas oferecem estruturas de muito baixo nível para serem de utilidade para explicar como utilizar o framework ou para compreender as colaborações que podem ser descrita através de contratos entre múltiplos participantes.

Os padrões de projeto podem ser de grande utilidade para aumentar o nível de abstração da descrição de um framework. A descrição do projeto em termos dos padrões de projeto envolvidos pode ajudar a compreender as caraterísticas das soluções providas pelo framework num nível de abstração muito maior que o possível a nível de classes. Cada padrão de projeto possui uma função e estrutura preestabelecidas que evita deduzir a função dos métodos e as classes no projeto a partir das interações. No entanto, os padrões atualmente reconhecidos são relativamente elementares, até óbvios na maioria dos casos, e variam muito tanto no nível de abstração quanto generalidade dos problemas que tratam. Adicionalmente, se considera-se que a estrutura de classes de uma aplicação não, necessariamente, respeitará a estrutura de classes sugerida pela definição do padrão, é necessário explicar detalhadamente a forma em que o padrão é refletido na hierarquia de classes. Lajoie e Keller propõem uma estratégia que combina a técnica de contratos com a descrição da hierarquia de classes [LAJ 94]. Segundo eles, um padrão de projeto é materializado através de uma micro-arquitetura de classes, a qual é descrita na sua estrutura por um contrato. Com esta documentação é possível compreender o fluxo de controle entre os participantes do contrato, com relação ao padrão de projeto que eles materializam. O padrão é considerado como uma descrição de maior nível de abstração que a micro-arquitetura, sendo o contrato a ponte entre a implementação das classes e o seu objetivo. Este enfoque, porém, apresenta o problema que a descrição da arquitetura independente do padrão obriga a permanente atualização da documentação, com os riscos de ficar desatualizada durante o desenvolvimento.

Este último aspecto apresenta um outro fator que contribui para a dificuldade da documentação de frameworks. Um framework, habitualmente é o resultado de várias iterações de projeto através das quais a estrutura inicial evolve a maiores níveis de abstração. Isto implica que a documentação de projeto deva ser atualizada em função dessas mudanças, as quais podem produzir-se dentro de prazos curtos quando o framework está nas primeiras etapas de desenvolvimento.

3.4 O papel dos Exemplos

Os exemplos são um elemento fundamental para facilitar a compreensão e o aprendizado de conceitos complexos. A partir de exemplos concretos, conceitos abstratos podem ser compreendidos e explicados. A abstração é um processo de síntese que se realiza a partir do conhecimento de casos concretos da realidade que uma pessoa percebe.

Lange e Moher estudaram o comportamento de programadores Smalltalk. Seu estudo demonstra que a reutilização efetiva depende essencialmente do conhecimento que o programador possui

acerca de casos semelhantes existentes na biblioteca de classes. No caso geral, a produção de código novo é evitada quando existe a possibilidade de modificar código de algum exemplo existente [LAN 89].

Os exemplos de aplicações construídas com um dado framework tem um papel de especial relevância dentro das diferentes fontes que podem contribuir para facilitar a compreensão desse framework. A análise da forma em que um framework é utilizado para construir aplicações é de grande ajuda para compreender como as abstrações providas são instanciadas para formar uma aplicação concreta. A análise de exemplos, em muitas circunstâncias, permite reduzir a necessidade de interpretar documentação extensa e complexa, bem como completar o modelo mental que o usuário cria através dessa documentação.

A documentação interativa provida pelo ambiente Smalltalk (*cookbook*) é um *exemplo* interessante que demonstra a utilidade dos exemplos para facilitar a utilização de uma biblioteca de componentes prontos baseada em um framework. Esta documentação descreve a forma de utilizar os componentes da biblioteca, detalhando os passos a serem seguidos para incorporar uma dada funcionalidade a uma aplicação, modificar os atributos do componente, e como combiná-lo com outros. Por exemplo, a seguir apresenta-se uma copia da página do manual interativo que descreve o processo documentado para incorporar uma interface de anotador a uma aplicação (o código ressaltado com moldura e itálico são exemplos mostrados em uma janela independente)

ADDING A NOTEBOOK

STRATEGY

A notebook is a powerful navigational widget. At its simplest, as shown here, it provides a list in the form of index tabs. When the user selects an index tab, the effect is the same as selecting an item in a conventional list – in fact, both a list and a notebook's tabs use a **SelectionInList** as their model. A notebook can be used in many of the same situations in which a list or a menu might be used, though its richer set of capabilities (such as minor keys) extend its range of uses.

A notebook also contains a subcanvas. This subcanvas can be used to display a different interface for each index tab or, as in this simple example, the same interface. In **Notebook1Example**, the subcanvas contains a list widget, and the list is changed each time an index tab is selected.

BASIC STEPS

Online example: **Notebook1Example**

1 Use a Palette to paint a notebook widget on your canvas.

2 In the notebook's Major property (on the Basics page of the Property tool), enter the name of the method (majorKeys) that returns a SelectionInList containing the labels for the index tabs.

3 In the notebook's ID property, enter an identifying name (pageHolder).

4 Create the canvas (listSpec) that is to be displayed inside the notebook. (In this example, a single canvas is used.)

5 Use a System Browser or the canvas's method->define command to create the instance variable (majorKeys) and accessing method (majorKeys) for the notebook's list of index labels. Initialize the variable, either in the accessing method or in an initialize method (as in the example), with a SelectionInList containing either strings or associations.

```
majorKeys                                        "Step 5"
    ^majorKeys
```

6 Similarly, create any variables and methods needed by the subcanvas. (In the example, these are the classNames variable, the classNames method, and the initialize method.)

```
classNames                                              "Step 6"
    ^classNames
```

7 In the initialize method, use an onChangeSend:to: message to arrange for the notebook to send a message (changedLetter) to the application model when the user selects an index tab.

```
changedLetter                                           "Step 8"

    | chosenLetter list |
    chosenLetter := self majorKeys selection last.
    list := Smalltalk classNames select: [ :name | name first == chosenLetter].
    self classNames list: list.
```

8 Create the change message (changedLetter) in which the subcanvas is updated based on the index tab that has been selected. (In the example, the classNames list is updated with classes beginning with the letter on the index tab.)

9 Create a postOpenWith: method. In this method, first get the notebook from the application model's builder, using the notebook's ID (pageHolder). Then install the subcanvas by sending a client:spec: message to the notebook. The first argument is the subapplication's application model (in the example, self). The second argument is the name of the spec method (listSpec) that defines the desired canvas.

```
postOpenWith: aBuilder                                  "Step 9"
    (aBuilder componentAt: #pageHolder) widget
        client: self
        spec: #listSpec.
    majorKeys selectionIndex: 1.
```

Estas explicações são complementadas por classes com exemplos típicos de instanciação do componente, os quais podem ser copiados e modificados para cada aplicação particular. Deste modo, um usuário pode utilizar o framework de interfaces, por exemplo, sem necessidade de compreender em detalhe seu projeto. No entanto, para aplicações que excedem os usos habituais previstos nos exemplos para um dado tipo de componente, é necessário compreender o seu projeto interno, o qual implica no problema descrito nas seções precedentes. Assim, ferramentas de apoio para análise destes exemplos se convertem em um complemento muito importante para facilitar a exploração de um framework em todo o seu potencial.

3.4.1 Livros de projeto e documentação

A metáfora do livro ou manual, tal como utilizada pelos *cookbooks* ou, mais avançados ainda, os ***technology books*** [ARA 93], representa uma metáfora muito vantajosa para a organização de informação de projeto a ser reutilizada. Um *technology book* organiza o conhecimento disponível acerca de uma classe de problemas (por exemplo, processamento de sinais), utilizando uma base de dados orientada a objetos. O conhecimento de projeto, análise e código é contido em objetos relacionados por relacionamentos de semântica bem definida. Além das relações clássicas de agregação e especialização, são definidas relações do tipo IBIS [CON 91] para suportar a modelagem de deliberações, e relações arbitrárias utilizadas para compor justificativas. Estes relacionamentos são utilizados para navegar através do *technology book* e raciocinar acerca da informação nele contida, utilizando técnicas de hipertexto.

Tipicamente um *technology book* inclui modelos analíticos de classes de soluções, modelos de projeto para estes modelos analíticos, e implementação destes projetos. Além desta informação, contém as explicações que justificam as implementações em termos do projeto bem como o

projeto em termos da análise correspondente. Com isto um projetista pode percorrer um projeto desde os requisitos até as possíveis implementações, fazendo consultas acerca de porque foram tomadas certas decisões em cada nível. O desenvolvimento de aplicações envolve vários *technology books* relativos aos diferentes aspectos tecnológicos que a aplicação abrange. Os projetistas contam com o suporte automatizado para navegar sobre esta informação e recuperar componentes de software adequados aos requisitos da aplicação sendo desenvolvida.

A construção de livros de documentação, entretanto, é uma tarefa custosa, ainda que dispondo de suporte eletrônico para sua manipulação. Assim, ferramentas de análise de exemplos, podem contribuir em muito para facilitar a tarefa de construção de livros, ajudando a documentar a análise de exemplos específicos, bem como sua generalização implementada por um dado framework. A organização da informação em termos de livro habilita a combinação de informação gráfica e textual, permitindo a utilização de atributos de texto para destacar visualmente exemplos de código, os quais podem se relacionar com sua representação gráfica, e explicações ou comentários introduzidos pelo utilizador do framework.

4 Ferramentas de Apoio à Compreensão e Técnicas de Suporte

No capítulo anterior discutiram-se os diferentes aspectos que contribuem para tornar difícil a compreensão de frameworks, desde o ponto de vista da complexidade inerente dos mecanismos da orientação a objetos, até as limitações das técnicas de documentação existentes. Como conclusão surge que parte dessa dificuldade pode ser reduzida através de ferramentas de software de apoio para a análise de exemplos de como um framework pode ser utilizado para construir aplicações.

Ferramentas de software para apoio à compreensão de programas são de grande importância para ajudar a reduzir a complexidade inerente do processo de compreensão. Estas ferramentas auxiliam o programador na construção de um modelo mental do programa, através de mecanismos de análise, exploração e visualização da informação em diferentes níveis de abstração. Freqüentemente, fornecem diferentes representações visuais que sintetizam propriedades relevantes do artefato analisado, bem como mecanismos de filtragem, organização e abstração de informação que permitem ao usuário explorar o artefato desde diferentes perspectivas. As características destes mecanismos, referidos genericamente como técnicas de suporte, determinam a eficácia de uma ferramenta para adequar-se a diferentes atividades e domínios de aplicação [TIL 96].

Neste capítulo analisam-se as diferentes técnicas de suporte que podem contribuir para facilitar o processo de compreensão de frameworks. A discussão centra-se, em primeiro lugar, na identificação das atividades típicas que definem um modelo de domínio para as ferramentas de análise de programas. Com base nesta caracterização, cada uma das técnicas específicas são analisadas tomando em consideração as abordagens existentes nas áreas de engenharia reversa e visualização de programas, os desenvolvimentos existentes na literatura para apoio à compreensão de software orientado a objetos e sua aplicabilidade para suportar a compreensão de frameworks.

4.1 Estratégias de Compreensão

A partir dos trabalhos de Solloway e Brooks, dentre outros, nos inícios da década dos 80, o estudo do comportamento dos programadores durante o processo de manutenção tem se convertido em uma área de intensa pesquisa. Compreender como os programadores conduzem o processo de compreensão de um dado programa, as fontes de conhecimento que utilizam e como adquirem conhecimento acerca do programa estudado é essencial para fornecer suporte automatizado para apoio nesta tarefa.

A compreensão de um programa pode ser caracterizada como um processo através do qual um programador constrói um *modelo mental* desse programa. Um modelo mental é uma representação conceitual que um programador tem do artefato em consideração, constituída por diferentes fontes de conhecimento. O conhecimento geral de estruturas de programação, problemas, algoritmos, planos, etc., é uma fonte que permite conduzir o processo de compreensão por caminhos produtivos e facilita a compreensão de novos artefatos. Também o conhecimento de problemas semelhantes permite a um programador experiente compreender, mais rapidamente, um novo artefato. Ou seja, o conhecimento do domínio de aplicação do programa é um fator de grande importância para facilitar a compreensão de um sistema específico [PEN 87]. Pennington denomina este conhecimento *modelo de situação*. Este modelo descreve o domínio de problemas em um nível de abstração maior que o modelo de programação. O processo de compreensão é, então, o resultado de mapeamento e coordenação entre o modelo de programação e o modelo de situação. Este conhecimento é utilizado para reconhecer estruturas abstratas a partir do conhecimento adquirido durante o processo de análise, tal como estrutura de controle do artefato, funcionalidade e arquitetura global [MAY 95]. Através do modelo mental obtido no processo, o programador é capaz de decidir como o artefato deveria ser modificado para satisfazer os novos requisitos.

Recentemente, Mayrhauser e Vans [MAY 95] classificaram e compararam as diferentes estratégias utilizadas por programadores para conduzir o processo de compreensão. Basicamente, o processo de compreensão é realizado segundo as estratégias *bottom-up*, *top-down* ou alguma combinação destas estratégias. Na estratégia *bottom-up* o programador parte do código fonte do sistema para reconstruir o seu projeto de alto nível. O processo de compreensão é realizado através de vários passos que envolvem agrupamento de funcionalidade e reconhecimento de conceitos de domínio, entre outros. Esta estratégia pressupõe um desconhecimento total do programa considerado. Assim, um primeiro passo habitual é a compreensão da estrutura de controle do programa, para depois avançar na compreensão de construções de maior nível de abstração. Uma estratégia *top-down*, por sua vez, requer um conhecimento inicial da funcionalidade do sistema, a partir do qual, procede-se à identificação dos componentes individuais responsáveis pela realização de cada tarefa do sistema. Esta estratégia adequa-se melhor para aqueles casos nos quais o programador possui experiência prévia na compreensão de sistema semelhantes. Na estratégia iterativa, o processo é considerado como um processo contínuo, durante o qual, a implementação do programa é analisada e comparada com o modelo conceitual do programador acerca da forma em que o problema deveria ser resolvido [DEP 94]. Esta estratégia representa um processo de refinamento e contrastação de hipóteses acerca do funcionamento do sistema, até que pode ser explicado em termos de um conjunto consistente de hipóteses [BRO 83].

A conclusão mais relevante destes estudos é que um programador não se baseia exclusivamente em uma única estratégia. No caso geral, diferentes estratégias são utilizadas em função do tipo, magnitude e conhecimento *a-priori* de cada problema a ser compreendido. Entretanto, experimentos tem demonstrado que programadores experientes utilizam basicamente uma estratégia *top-down*, orientada por planos que definem as suas expectativas acerca de como o programa deveria funcionar; mas, quando essas expectativas não são satisfeitas uma estratégia *bottom-up* é adotada.

Esta conclusão é importante para o projeto de ferramentas de apoio. Uma ferramenta adequada para suportar eficientemente o processo de compreensão deve ser o suficientemente *flexível* para adaptar-se a cada potencial estratégia utilizada pelo usuário. Esta flexibilidade é ainda mais relevante no caso de programas orientados a objetos, e particularmente dos frameworks, como é discutido a seguir.

4.1.1 Compreensão orientada pela arquitetura

No caso da compreensão de frameworks, vários aspectos devem ser considerados para definir os requisitos que uma ferramenta deve satisfazer para atender adequadamente o processo da compreensão de um dado framework.

No capítulo 2, definiu-se o conceito de framework como, essencialmente, a implementação, em termos de classes de uma arquitetura genérica para um domínio de aplicação. Um conhecimento prévio do domínio de aplicação é, sem dúvida, de grande importância para facilitar a compreensão de um dado framework. Através do conhecimento geral do domínio, um programador pode compreender a organização geral de conceitos, ou, mais especificamente, o modelo de domínio que o framework implementa. Por outro lado, também é necessário ter em consideração que o objetivo do desenvolvimento de um framework é permitir a reutilização, por parte dos usuários do framework, do conhecimento de domínio que o projetista possui. Assim, é razoável esperar que os usuários do framework não necessitem possuir um conhecimento profundo do domínio de aplicação, mas só o conhecimento necessário da funcionalidade da aplicação que deseja-se implementar. Idealmente, para ser realmente de utilidade, um framework deve permitir ao usuário a implementação de aplicações partindo do conhecimento da funcionalidade que as classes abstratas deixam para ser implementada por subclasses. Assim, um primeiro passo razoável no processo de compreender um framework é prover o usuário com mecanismos que lhe permitam construir um modelo mental inicial da estrutura e comportamento da arquitetura implementada por esse framework.

Através de visualizações centradas na arquitetura, uma ferramenta pode prover um mecanismo estruturado para compreender um dado framework, começando em um alto nível de abstração, provido pela visualização dos aspectos essenciais da arquitetura implementada por classes

abstratas. Visualizações que ressaltem componentes abstratos, a categoria de seus métodos (*abstratos*, *template*, *hook* e *base*) e como eles se relacionam através da troca de mensagens, podem prover uma primeira aproximação muito útil para compreender a organização global do framework. A visualização destes aspectos é particularmente útil para compreender quais componentes devem ser especializados e quais métodos devem ser implementados.

A dicotomia entre o modelo estático e o modelo dinâmico dos programas orientados a objetos, entretanto, impõe a necessidade de fornecer mecanismos que facilitem para um programador a construção do mapeamento entre a organização dinâmica de instâncias e a organização estática de classes. Uma compreensão no nível arquitetônico permite alcançar uma compreensão global das características essenciais de um framework, mas podem existir aspectos que não podem ser completamente compreendidos através de uma representação arquitetônica. Em alguns casos é necessário analisar em detalhe a configuração interna de instâncias envolvidas em funções específicas do framework. Estados internos de instâncias também podem determinar comportamentos muito diferentes que precisam ser compreendidos. Portanto, mecanismos para visualizar e inspecionar as configurações de instâncias, em determinados pontos da execução dos exemplos, são também necessários para facilitar o processo de compreensão de um dado framework.

A implementação destes mecanismos, porém, pode ser problemática. Manter informação acerca das instâncias para ser analisada pelo usuário é usualmente um problema difícil de resolver eficientemente se considera-se que, para algumas aplicações, o número de instâncias criadas em uma simples execução pode alcançar facilmente várias centenas de milhares. Algumas propostas existentes na literatura utilizam animações em tempo real para aliviar este problema, mas este tipo de animações resulta geralmente confusa quando as instâncias são criadas e destruídas muito rapidamente durante a execução da aplicação. Estes problemas acentuam-se em linguagens como Smalltalk, devido à coleta automática de instâncias não referenciadas. Neste caso não é possível conhecer (pelo menos com um custo computacional razoável) quando uma dada instância não é mais referenciada, de modo que a quantidade de espaço necessário para manter a informação de instâncias poderia impor um custo intolerável em termos de requisitos de memória.

Uma abordagem alternativa para resolver estas limitações é a provisão de um suporte que habilite a exploração dos aspectos arquitetônicos e de organização de instâncias segundo uma estratégia iterativa. Uma vez que os aspectos globais da organização do framework tem sido analisados e compreendidos, então é possível avançar na análise detalhada do comportamento das instâncias envolvidas em colaborações específicas. Através da representação visual da arquitetura, o usuário pode selecionar fazer a inspeção detalhada daquelas instâncias envolvidas em colaborações específicas ou estabelecer pontos de corte em métodos determinados que requerem de um estudo mais detalhado. Executando repetidamente a mesma aplicação (ou continuando a mesma execução) a ferramenta pode coletar informação das instâncias selecionadas e suportar a visualização e análise detalhada daqueles aspectos selecionados pelo usuário. Novas classes, ou subclasses, podem ser analisadas caso seja necessário aumentar ou refinar o grau de compreensão para desenvolver uma dada aplicação.

Esta estratégia iterativa pode ser considerada como uma estratégia *inside-out* [BAT 92], para distinguí-la das abordagens *top-down* e *bottom-up* já descritas. Sob esta estratégia, um usuário pode estudar gradualmente o comportamento e estrutura de um dado conjunto de classes que surgem, em primeiro lugar, como relevantes (talvez baseado no conhecimento prévio que possua do domínio) e enriquecer, gradualmente, o modelo mental com o estudo detalhado de instâncias e classes adicionais. Um mecanismo desta natureza, porém, requer técnicas de suporte que permitam adequar a ferramenta dinamicamente para realizar a análise nos diferentes níveis de abstração envolvidos. A seguir, discutem-se as abordagens existentes à luz destes requisitos e descrevem-se sinteticamente as abordagens adotadas neste trabalho.

FIGURA 4.1- Atividades básicas das ferramentas de análise de programas

4.2 Modelo Essencial do Domínio

Os sistemas de engenharia reversa e de visualização de software representam as duas principais linhas de trabalho que mais tem contribuido para o desenvolvimento de técnicas de suporte para a compreensão de software.

Engenharia reversa é a atividade através da qual, informação relevante acerca de um programa ou sistema é identificada, seus relacionamentos descobertos e abstrações são geradas [CHI 90]. As ferramentas de suporte para esta atividade objetivam prover mecanismos automáticos para a extração de informação de um programa a partir da análise do código e deduzir ou reconhecer abstrações estruturais e comportamentais não representadas diretamente nele.

Os sistemas de visualização de software, por sua vez, objetivam prover mecanismos para construir representações visuais da informação do programa, através das quais o usuário pode analisar e compreender caracteristicas de estrutura e funcionamento em um maior nível de abstração que o código.

Dentro da ampla gama de sistemas de visualização de software existentes, os sistemas de animação de algoritmos, como por exemplo BALSA [BRO 84], BALSA II [BRO 88], TANGO [STA 90], etc., representam uma classe particularmente relevante. Estes sistemas fornecem o suporte para a construção de animações do funcionamento de um programa, as quais são especialmente adequadas para ajudar na compreensão da forma em que algoritmos e programas complexos se comportam em tempo de execução.

Essencialmente, ambos tipos de sistema diferenciam-se na ênfase colocada nos aspectos de análise e geração de abstrações. Os sistemas de animação centram-se na provisão de mecanismos para construção de visualizações dinâmicas a partir de relacionamentos programa-visualização definidos pelo usuário, enquanto as ferramentas de engenharia reversa colocam a ênfase na provisão de mecanismos *automáticos* de análise de código e reconhecimento de padrões. Os sistemas de animação tendem a ser mais gerais do ponto do vista da sua adequação a diferentes linguagens de programação, enquanto os sistemas de engenharia reversa tendem a ser dependentes da linguagem. Entretanto, os dois tipos podem ser caracterizados por uma seqüência básica de quatro atividades que abrangem (FIGURA 4.1):

- Captura de informação do programa alvo
- Representação dessa informação em termos adequados para sua análise automática ou manual
- Análise da informação e reconhecimento de abstrações

- Apresentação da informação ao usuário.

Cada uma destas atividades representa, por si só, uma complexa área de problemas. Diferentes ferramentas requerem soluções particulares dependendo dos objetivos ou tarefas para as quais são projetadas. Por esta razão, diferentes técnicas tem sido propostas na literatura e adotadas em combinações particulares por distintas ferramentas.

A seguir descrevem-se as soluções mais relevantes adotadas pelas ferramentas para a análise de programas orientados a objetos existentes na literatura, discutindo a adequação destas ferramentas para apoiar a compreensão de frameworks.

4.3 Captura de Informação

A captura de informação do programa alvo é o primeiro passo para gerar a informação necessária para a análise ou visualização desse programa. As técnicas de análise utilizadas habitualmente para implementar este processo podem classificar-se como estáticas e dinâmicas.

A análise estática é a técnica predominante utilizada pelas ferramentas de engenharia reversa. A análise estática consiste, essencialmente, no reconhecimento sintático (*parsing*) do código fonte, a partir do qual uma representação abstrata do programa é gerada. A representação mais comum são árvores de sintaxe abstrata, as quais contêm uma grande quantidade de unidades de informação sintática de granularidade fina em conjunto com as diferentes dependências entre essas unidades. Sobre esta representação, diferentes funções de análise são aplicadas para gerar ou recuperar abstrações de maior nível que o código, descobrir relacionamentos entre essas abstrações e produzir diferentes representações visuais do programa analisado.

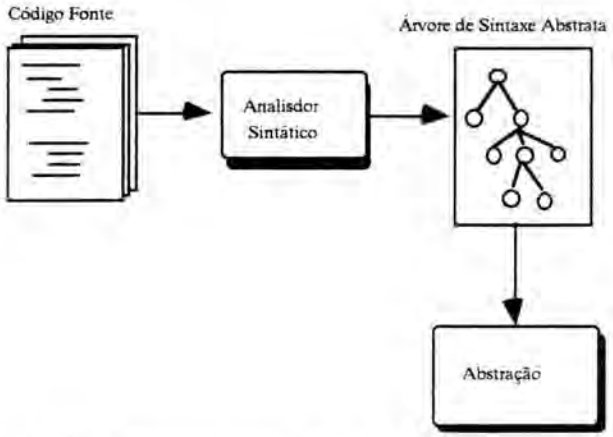


FIGURA 4.2. Estrutura típica de ferramentas baseadas em análise estática

Este tipo de técnica tem sido utilizado com sucesso em sistemas de engenharia reversa de programas procedimentais tradicionais. Ferramentas de apoio ao desenvolvimento de frameworks orientados a objetos, como *Software Refactory* [OPD 92] também utilizam técnicas de análise estática. Esta ferramenta suporta o processo de restruturação de um framework programado em C++, partindo da análise estática de aplicações construídas com esse framework. *Software Refactory* implementa em forma semi-automática várias refatorações de classes, como por exemplo: criação de superclasses abstratas a partir de classes concretas, divisão de uma classe em diferentes subclasses e substituição de relacionamentos de herança por agregação, entre as mais importantes. Estes processos implicam na reorganização das hierarquias de classes, o deslocamento de métodos e a criação de novos métodos. As mudanças no código necessárias, no entanto, não são simples, pois o comportamento comum pode conter aspectos específicos a cada implementação. A semântica de muitas construções não pode ser inferida através da análise léxica e estrutural e portanto é necessária a intervenção do projetista para

determinar quando uma dada transformação deve ser aplicada ou não. Este aspecto, porém, é desejável, pois o processo de refatoração é uma atividade a ser realizada pelo projetista do framework, o qual conhece as razões pelas quais algumas construções de projeto foram implementadas de uma maneira específica.

As técnicas de análise estática, úteis sem duvida, não são, entretanto, suficientes para suportar a compreensão de software orientado a objetos. Geralmente, através da análise do código das classes não é simples determinar com certeza os padrões de colaboração que se materializam em tempo de execução [LAN 95]. Uma outra limitação é o caso de linguagens não tipadas, como por exemplo Smalltalk. Neste caso, a ausência de tipos impede que se determine estaticamente, pelo menos na maioria dos casos, qual a classe da instância receptora de uma mensagem. Deste modo, não é possivel recuperar os relacionamentos entre classes que definem o projeto de um programa ou de um framework.

As técnicas de análise dinâmica consistem na captura de informação acerca do comportamento do programa em tempo de execução. A informação de tempo de execução pode ser muito valiosa para compreender aspectos do comportamento de um programa que não podem ser deduzidos estaticamente. Ferramentas de medição de desempenho (*profilers*), por exemplo, são de utilidade para identificar aquelas porções de um programa nas quais concentra-se a maior carga de computação e, portanto, alvos de otimização.

O mecanismo utilizado pela maior parte das ferramentas atuais para capturar a informação dinâmica é a instrumentação do código fonte para o anúncio de eventos de interesse ou transições de estados do programa. Um evento de interesse denota uma ação realizada pelo programa, como por exemplo ativação ou retorno de um procedimento ou modificação do estado de uma variável. A execução de um programa pode ser representada como uma seqüência de eventos, a qual é utilizada para produzir uma representação, geralmente gráfica, da execução. A noção de evento é especialmente adequada para construir ferramentas de análise dinâmica porque suporta a captura de informação em diferentes níveis de abstração. Um evento pode representar ações elementares de um programa, como envio de uma mensagem, até a execução de uma função completa do sistema.

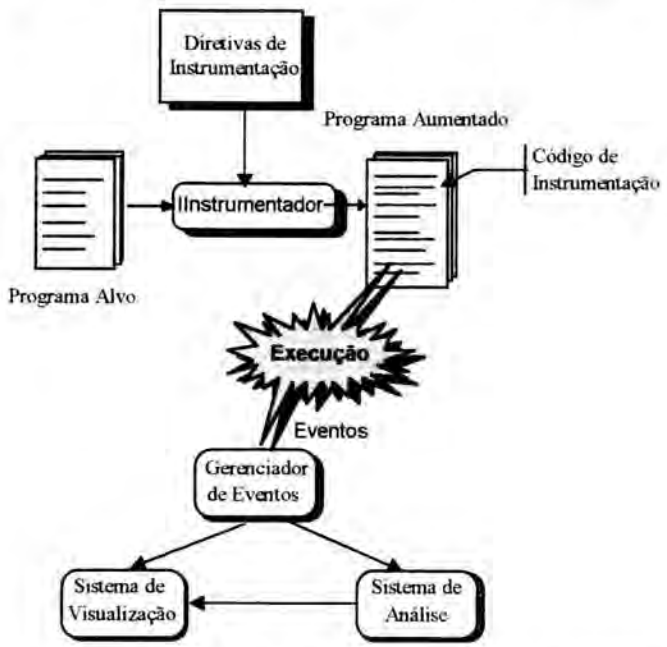


FIGURA 4.3. Estrutura típica de ferramentas baseadas em análise dinâmica

O mecanismo de instrumentação consiste na ampliação, manual ou automática, do código da aplicação com anotações que invocam procedimentos de animação ou geram a informação relativa ao evento que representam (FIGURA 4.3). Esta abordagem é a mais utilizada pelas ferramentas de análise de programas orientados a objetos mais significativas desenvolvidas na atualidade, como

ObjectVisualizer [DEP 93] [DEP 94], *ProgramExplorer* [LAN 95],*VizBug++* [STA 94] e *BEE++* [BRU 93] para análise de programas C++.

ObjectVisualizer e *ProgramExplorer* utilizam um mecanismo de instrumentação automática baseada em *scripts*, os quais definem as ações do programa a serem interceptadas e os eventos que denotam. O instrumentador aumenta o código do programa com código de instrumentação encarregado de gerar informação acerca desses eventos, a qual é processada localmente ou transmitida através de uma rede. Os eventos gerados pelo código de instrumentação são recebidos por um gerenciador de eventos encarregado de registrá-los e comunicá-los ao sistema de visualização. VizBug++ e BEE++ utilizam mecanismos semelhantes mas a anotação do código deve ser realizada manualmente, o que torna mais complicada a sua utilização.

A principal limitação destes mecanismos é a necessidade de modificar o código da aplicação para incorporar as anotações. Adicionalmente, informação estática relevante para a compreensão da estrutura e projeto da aplicação, como por exemplo redefinição de métodos, deve ser capturada através da análise estática da aplicação. O ProgramExplorer combina as duas técnicas utilizando informação adicional gerada pelo compilador para produzir uma representação Prolog da estrutura estática do programa, e o mecanismo de anotação para gerar informação dinâmica (FIGURA 4.4). Esta estratégia permite a análise da estrutura e do comportamento dinâmico do programa em forma integrada, mas a solução depende da implementação específica do compilador utilizado.

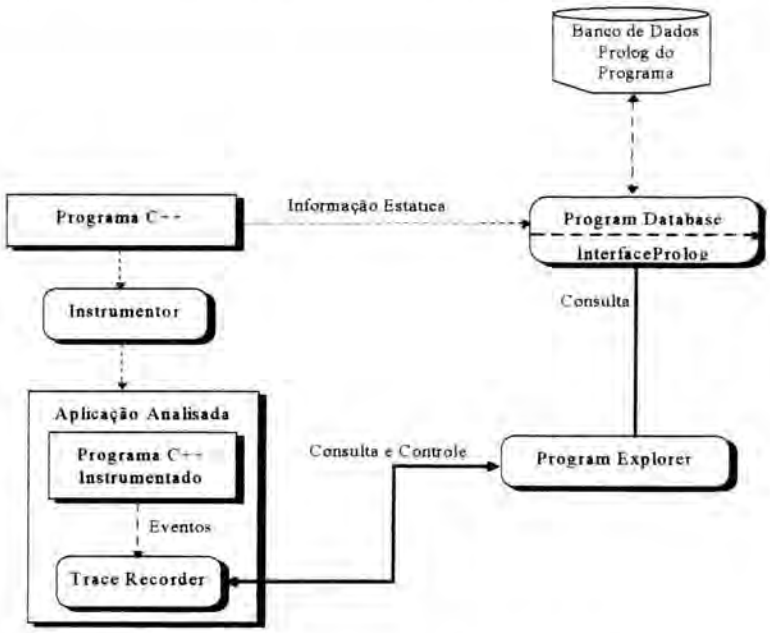


FIGURA 4.4 Estrutura geral do *Program Explorer*

4.3.1 Reflexão computacional

Reflexão é uma técnica alternativa para a coleta de informação de um programa. Reflexão é a capacidade de um sistema computacional para realizar computação sobre sua própria computação [MAE 87], ou seja, é capaz de analisar (refletir sobre) e modificar as entidades que definem seu próprio comportamento.

Esta capacidade implica na existência de uma representação das entidades que definem o comportamento do programa (funções, sentenças, dados, etc.), acessível em termos de dados. Esta representação é denominada *auto-representação*, pois é uma representação que o programa possui de si mesmo. Lisp é um exemplo deste tipo de representação. Linguagens reflexivas baseadas em Lisp, como 3-KRS por exemplo, caracterizam-se por serem implementadas por um *interpretador meta-circular*,

isto é, o próprio interpretador é implementado na mesma linguagem e acessível como um dado. A modificação da representação do interpretador produzirá a modificação do seu comportamento, modificando, em conseqüência, o comportamento do próprio programa.

A capacidade de modificar a auto-representação é denominada *efetuação*. A capacidade de inspecionar a auto-representação é denominada *introspeção*. Através de introspeção um programa pode raciocinar acerca de seu próprio estado e analisar seu comportamento [GAB 91].

A capacidade de introspeção de uma linguagem é um mecanismo essencial para facilitar a coleta de informação de um programa escrito nessa linguagem. Através da análise da auto-representação, tanto aspectos estáticos da estrutura do programa quanto aspectos dinâmicos de seu comportamento podem ser extraídos sem necessidade de ferramentas *ad-hoc* para tal fim.

No paradigma de orientação a objetos o conceito de *meta-objeto* foi introduzido por Patty Maes [MAE 87] como um mecanismo para suportar comportamento reflexivo em linguagens orientadas a objetos. Em uma linguagem orientada a objetos reflexiva baseada em meta-objetos o *comportamento computacional* de cada objeto que compõe uma aplicação é determinado por um meta-objeto que controla e define a operação do seu objeto.

Estas linguagens são suportadas por uma arquitetura em dois níveis: um nível base, que contém os objetos relacionados ao domínio de aplicação, e um meta-nível, onde residem os meta-objetos cujo domínio são os objetos do nível base. Basicamente o comportamento dos objetos no nível base é determinado pelos meta-objetos através do controle da ativação dos métodos. Assim, novos comportamentos podem ser acrescidos através da atuação sobre o processo de instanciação ou modificando a forma como os métodos são executados. Um meta-objeto, por exemplo, pode delegar a execução de um dado método a um outro objeto, ou ainda distribuir esta responsabilidade entre um conjunto de objetos.

A implementação de reflexão baseada em meta-objetos oferece um alto nível de flexibilidade para construir bibliotecas de comportamentos reflexivos, os quais podem ser *acrescentados* em forma *transparente* e *não intrusivamente* (a respeito do código fonte) a um sistema existente, tanto para estender a sua funcionalidade quanto para analisar a estrutura e comportamento de um programa.

Os meta-objetos podem colaborar entre si e serem organizados em uma arquitetura, denominada *meta-arquitetura*, que forneça o suporte para implementar complexas funcionalidades de análise dinâmica. Um enfoque baseado na utilização de meta-objetos, oferece a possibilidade de construir ferramentas *dinamicamente* adaptáveis para analisar o comportamento de aplicações através da especialização de meta-objetos. Diferentes meta-objetos podem ser associados dinamicamente com os objetos de uma aplicação a ser analisada, permitindo durante a mesma execução, seletivamente analisar aspectos específicos do seu comportamento (FIGURA 4.5). Esta capacidade é de grande

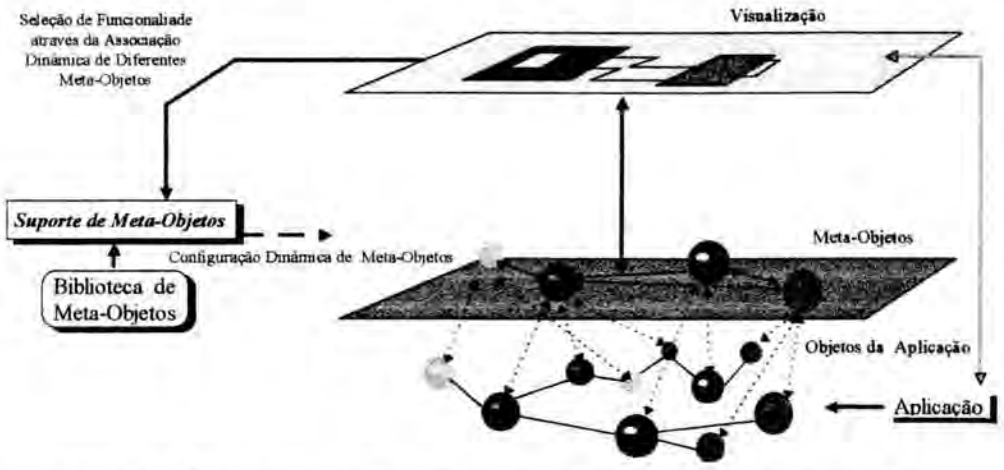


FIGURA 4.5- Arquitetura de configuração dinâmica baseada em meta-objetos

utilidade para suportar o enfoque orientado pela arquitetura antes descrito, pelas seguintes razões:

- ***Captura Arquitetônica***: Tomando em consideração que meta-objetos tem a capacidade de acessar a estrutura de classes e interceptar o comportamento dos objetos que compõem uma aplicação, é possível refletir as aplicações a serem analisadas sobre um conjunto de meta-objetos especialmente projetados para reconhecer componentes abstratos e seus refinamentos, categorias de métodos e fluxo de mensagens entre componentes, os quais fornecem a informação necessária para suportar visualizações arquitetônicas.
- ***Análise de instâncias:*** A análise do comportamento de instâncias específicas pode ser suportada por meta-objetos especialmente projetados para registrar informação acerca de instâncias envolvidas em seqüências específicas de passagem de mensagens ou para estabelecer pontos de corte em métodos ou mensagens. Estes meta-objetos podem ser *dinamicamente associados* com a aplicação analisada através da visualização da arquitetura gerada durante a primeira fase.

Apesar de existirem diversas implementações de domínio público [CHI 95][HIN 94], a utilização de técnicas de reflexão não tem sido ainda explorada para a construção de ferramentas de apoio à compreensão e reutilização. Esta abordagem é um dos pontos centrais deste trabalho e é discutida em profundidade no próximo capítulo.

4.4 Representação da Informação

A representação da informação coletada define tanto as capacidades da ferramenta quanto a sua escalabilidade. Para sistemas de mediana complexidade a quantidade de informação coletada pode ser enorme. Isto impõe a necessidade de organizá-la eficientemente em termos de espaço, facilidade de acesso e geração de abstrações.

As ferramentas baseadas em anúncio de eventos representam o espaço de informação através da sua associação com eventos de interesse. VizBug++, por exemplo, define o mapeamento entre entidades de programa e eventos mostrado na TABELA 4-1. *ObjectVisualizer* e *ProgramExplorer* definem um mapeamento semelhante.

TABELA 4-1- Espaço de Eventos Típico de Ferramentas de Análise Dinâmica

Classe	Evento	Parâmetros
Informação Estática	Definição de Classe	nome de classe
	Definição de Superclasse	nome de classe, nome de superclasse
	Definição de Método	nome de classe, nome do método, tipo do método, tipo do retorno, lista de argumentos
	Definição de Atributo	nome de classe, nome do atributo, tipo
	Definição de Função	nome, tipo do retorno, lista de argumentos
Informação de Objetos	Criação de Instância	nome de classe, nome de instância, ponteiro a self, nome de arquivo, número de linha
	Destruição de Instância	nome de classe, nome de instância, ponteiro a self, nome de arquivo, número de linha
Criação	Chamada a Construtor	nome de classe, ponteiro a self, nome de arquivo, número de linha, lista de argumentos
	Retorno de Construtor	nome de classe, ponteiro a self, nome de arquivo, número de linha
	Chamada de Destrutor	nome de classe, ponteiro a self, nome de arquivo, número de linha
	Retorno de Destrutor	nome de classe, ponteiro a self, nome de arquivo, número de linha

Classe	Evento	Parâmetros
Ativação	Chamada de Método	nome de classe, ponteiro a self, nome de arquivo, número de linha, lista de argumentos
	Retorno de Método	nome de classe, ponteiro a self, nome de arquivo, número de linha, retorno
	Chamada de Função	nome de classe, ponteiro a self, nome de arquivo, número de linha, lista de argumentos
	Retorno de Função	nome de classe, ponteiro a self, nome de arquivo, número de linha, retorno

Esta abordagem apresenta alguns inconvenientes. O conjunto adequado de eventos primitivos, para uma ferramenta, depende fortemente do nível de abstração necessário para cada atividade. Por exemplo, eventos denotando modificação de estado de variáveis são úteis para tarefas de depuração mas podem ser desnecessários para atividades de compreensão. Por esta razão, mecanismos para definir e adaptar eventos tornam-se um componente essencial para que uma ferramenta seja efetiva em uma ampla gama de aplicações que requerem a compreensão de um programa. O framework *BEE++* [BRU 93] adota uma representação orientada a objetos dos eventos, definindo hierarquias que podem ser especializadas para suportar a geração de eventos em diferentes níveis de abstração (por exemplo, conexão de cliente, controle remoto do programa, entrada de procedimento, acesso a variável, etc.) tanto predefinidos como definidos pelo usuário. Esta capacidade é de utilidade para o caso de ferramentas de depuração, mas, no caso dos frameworks, a caracterização do espaço em termos de eventos não é suficiente para apresentar informação estática relevante, como por exemplo, redefinição de métodos e categoria desses métodos. A solução habitual é simular eventos de definição de elementos do programa (classes, variáveis, métodos, etc.) que codificam informação estática. Estes eventos devem ser gerados antes da execução efetiva do programa.

Um outro problema é a necessidade de organizar de maneira eficiente os milhares de eventos gerados durante a execução do programa. Se estes eventos são processados de forma direta pelo sistema de visualização, podem produzir-se atualizações rápidas demais na visualização, inibindo a capacidade do programador para processar a informação mostrada pela ferramenta. Uma solução possível, provida pelo framework *BEE++*, é a utilização de um *buffer* de eventos que é controlado pelo programador, suspendendo a geração de novos eventos, enquanto o programador analisa detalhadamente a visualização. Esta solução torna mais complexo, evidentemente, o código de instrumentação. Sob esta abordagem o código de instrumentação deve ser capaz de suspender e reiniciar a execução do programa monitorado sob demanda do sistema de visualização, o qual impõe a necessidade de um protocolo de comunicação bidirecional entre a aplicação e a visualização.

O *Object Visualizer* utiliza uma estratégia de pre-processamento dos eventos. Diferentes eventos são combinados em "marcos de chamada" (*call frames*) armazenados em uma matriz, a qual contém a informação que será efetivamente utilizada para gerar visualizações. Uma entrada típica da matriz é descrita pela seguinte informação :

Classe Receptora ::Instância Receptora. Classe Implementadora :: Método Ativado

Com esta informação básica uma mensagem é descrita por uma entrada

Classe Origem ::Instância Origem. Classe Implementadora :: Método Origem
→ Classe Receptora ::Instância Receptora. Classe Implementadora ::Método Ativado

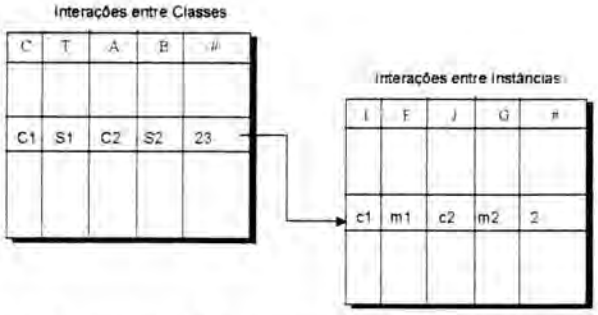


FIGURA 4.6- Representação de eventos utilizada pelo Object Visualizer

Os marcos são registrados sem repetição, mantendo uma contagem do número da sua ocorrência, organizados através uma tabela de *hashing* de dois níveis (FIGURA 4.6). Um primeiro nível contendo informação de relacionamentos entre classes e um segundo nível contendo informação acerca das instâncias envolvidas e os métodos ativados para essas interações.

O *Program Explorer* [LAN 95] utiliza uma representação Prolog da informação estática e dinâmica do programa analisado, a qual é criada e gerenciada por uma aplicação específica (*Program Database*).

A informação estática do programa é representada através do esquema de relações mostrado abaixo.

```
class(ClassName)
function(FunctionName, Type,VarList)
variable(VariableName, Storage,Type)
inherit(BaseClass, MemberName, Acess)
call(Caller, Callee)
use(User,Variable)
```

A informação dinâmica é descrita por eventos e mudanças de estado, representada segundo o esquema seguinte

```
create(SourceObject,SourceFunction,TargetObject,Class,Time)
destroy(SourceObject,SourceFunction,TargetObject,Time)
invoke(SourceObject,SourceFunction,TargetObject,TargetFunction,Time)
access(SourceObject,SourceFunction,TargetObject,Variable,Time)
value(Variable,Value,Time)
```

A principal vantagem desta representação é que permite a especificação da estrutura do programa ou framework em diversos níveis de abstração, através da definição de regras para filtragem de informação em tempo de execução.

4.4.1 Representação de hiperdocumento: Suporte à documentação

Sistemas de software complexos e, particularmente sistemas orientados a objetos, caracterizam-se por definir uma complexa rede de informação não linear. Esta rede pode ser composta por componentes, relacionamentos de herança, fluxo de dados e de controle, bem como elos de acesso a documentação existente e gerada durante o processo de análise e compreensão. Uma representação estrutural favorece a exploração desta informação através da navegação sobre o hiperespaço definido pelos nodos e elos que formam a rede.

Representações estruturais favorecem a integração de informação coletada com a documentação existente, bem como a geração de nova documentação. Este é um aspecto muito importante de ser contemplado. As ferramentas de visualização desenvolvidas para programas orientados a objetos, centram-se na provisão de mecanismos de suporte para atividades de compreensão e depuração, mas não contemplam sua integração com ambientes de desenvolvimento existentes, nem sua utilização para apoiar na documentação de frameworks. O desenvolvimento de frameworks é,

habitualmente, uma atividade iterativa que implica no desenvolvimento gradual do projeto genérico representado por classes abstratas. Este processo iterativo conduz ao problema de documentação desatualizada discutido no capítulo 3. Ferramentas como a *Software Refactory* [OPD 92], fornecem um suporte valioso para a manipulação e restruturação do código, mas não fornecem suporte nenhum para documentar o resultado das refatorações realizadas.

Do ponto de vista de um usuário ocasional de um framework, uma ferramenta deve, idealmente, fornecer mecanismos que habilitem esse usuário a utilizar o framework com um mínimo esforço de aprendizado de utilização da ferramenta. Neste caso, a documentação da informação obtida durante o processo de compreensão não é tão relevante. Do ponto de vista de uma organização, na qual um framework pode ser reutilizado muitas vezes, é conveniente que o processo de compreensão realizado por um usuário do framework sirva para enriquecer a documentação existente e seja reutilizado para reduzir os esforços necessários para o desenvolvimento de uma aplicação posterior. Neste ponto as ferramentas de visualização falham sistematicamente na provisão de um suporte adequado para esta tarefa.

Considerando a documentação eletrônica baseada em livros de projeto discutida no capítulo anterior, uma representação da informação coletada em termos de um modelo de hipertexto aparece com uma alternativa adequada. Esta representação habilita a organização de informação do framework e sua integração tanto com documentação existente quanto com a criada ao longo do processo de compreensão e utilização do framework.

Dentre as diversas alternativas de representação desta rede de informação, representações agregadas permitem associar informação altamente relacionada em nodos mais abstratos que evitam a necessidade de criar elos para apresentar esses relacionamentos. Este tipo de representações, como por exemplo o modelo de contextos aninhados (*Nested Context Model*) [CAS 91], permitem agrupar nodos relacionados, de acordo com algum critério, em objetos agregados ou compostos que podem ser tratados como uma unidade. Esta representação é de grande utilidade para representar, por exemplo, hierarquias de classes como um conjunto de nodos aninhados, bem como, representar diretamente abstrações compostas como subsistemas ou grupos colaborativos, as quais podem ser mapeadas diretamente a representações gráficas estruturais.

Neste trabalho adota-se uma representação baseada no modelo PROMETO[ORT 95], o qual é uma extensão orientada a objetos do modelo de contextos, especialmente desenvolvida para representar documentos estruturados de software.

4.5 Apresentação de Informação

A utilização de técnicas visuais é um componente essencial para facilitar o processo de compreensão de sistemas complexos. Basicamente, a visualização é um processo de conversão de dados em dimensões perceptuais, com o objetivo de facilitar a compreensão desses dados [ROG 93].

Técnicas de visualização, gráficas e textuais, permitem concentrar maior densidade de informação em uma apresentação e explorar a capacidade humana para análise de informação espacial e interpretação de cores. Também favorecem a construção de animações que facilitam o processo de compreensão de comportamento dinâmico, o qual não pode ser sinteticamente representado por notações textuais.

Independente do mecanismo de captura, a grande quantidade de dados coletados para compreender sistemas complexos requer mecanismos para visualizar e ressaltar aspectos relevantes dessa informação. Grandes quantidades de dados podem facilmente inibir a capacidade humana para assimilá-los e construir um modelo mental adequado [MIL 56]. Por esta razão, as técnicas de visualização providas são um fator chave para uma ferramenta apoiar eficazmente à compreensão de programas. Deste modo, o objetivo principal de uma visualização é condensar a maior quantidade possível de informação em uma mesma apresentação. Uma visualização compacta facilita o processamento imediato da informação transmitida [SIL 96].

A utilização de técnicas de visualização suportadas por computador introduz a capacidade de interação do usuário com a informação apresentada. O usuário pode, alem de visualizar

os dados, interagir e realizar mudanças na apresentação para, por exemplo, mudar o nível de abstração dos dados ou selecionar dados relevantes a seu interesse [JER 96]. Entretanto, é necessário tomar em consideração que o propósito de uma visualização é oferecer um veículo para a obtenção de conhecimento acerca da realidade sendo visualizada. Assim, um usuário deve ser capaz de observar os dados utilizando a técnica adequada para facilitar a compreensão desses dados. Segundo Andrienko, a seleção da técnica de visualização apropriada depende de vários fatores [AND 97], como por exemplo, as características dos dados e seus relacionamentos, os objetivos perseguidos pelo usuário e fatores de percepção humana.

Os sistemas de visualização de informação, e particularmente os sistemas de visualização de software, surgiram como um meio de fornecer suporte para o processo de compreensão de informação, através de facilidades para a construção de diferentes técnicas de visualização e de interação do usuário com essas visualizações. Neste contexto, Fishkin considera que as ferramentas de manipulação direta para exploração de relacionamentos entre os dados visualizados são um componente essencial de qualquer sistema de visualização [FIS 95]

A partir destas observações, um sistema de visualização pode ser caracterizado pela provisão de suporte para os aspectos seguintes:

- *Apresentação e Codificação visual da informação*: é a característica que define os sistemas de visualização. Uma representação visual (ou imagem) possibilita que o usuário (seja leitor ou explorador) possa observar a informação que tenta compreender através de representações visuais compacta que ressaltam aspectos relevantes da informação analisada.
- *Navegação*: é a técnica que permite que o usuário possa percorrer o espaço de informação visualizado. A navegação habilita o usuário a observar distintos aspectos dos dados, ver a informação desde diferentes perspectivas, observar dados com maior nível de detalhe, etc.
- *Interação*: Esta característica permite que o usuário possa interagir com a representação da informação que está visualizando. Esta interação implica na capacidade de realizar mudanças na forma que são apresentados os dados, modificar aspectos dos mesmos, selecionar um subconjunto de interesse (*focus*), etc.
- *Relacionamento com os dados*: O sistema de visualização deve estar vinculado, de alguma forma, com os dados visualizados. Nos sistemas de visualização de informação, estes dados se encontram, geralmente, armazenados em um banco de dados, e, na maioria dos casos, trata-se com grandes volumes de informação.

4.5.1 Apresentação e codificação visual

Do ponto do vista da visualização de um sistema de software, dois aspectos principais estão envolvidos na construção de visualizações: representação das entidades do programa sendo visualizado e a codificação de informação adicional relativa a essas entidades através de atributos visuais.

Roman e Cox identificam três amplas categorias nas quais podem classificar-se os diferentes meios de codificação de informação: objetos visuais e seus atributos, relacionamentos visuais entre esses objetos e eventos visuais [ROM 93].

Os objetos visuais são utilizados, geralmente, para codificar aspectos concretos do programa. Exemplos simples de objetos visuais são os objetos geométricos que compõem as notações habituais de projeto, como retângulos círculos, etc. Estes objetos podem ser combinados para formarem objetos complexos aos quais associam-se semânticas específicas. Atributos como cor, tamanho e fonte de texto, são utilizados para aumentar a semântica da apresentação, distinguir casos específicos da mesma abstração e focalizar a atenção do usuário em pontos determinados. Os relacionamentos visuais representam relacionamentos entre os objetos visuais, como por exemplo, informação estrutural do

programa que é representada por estruturas geométricas análogas. O exemplo típico destes relacionamentos é a representação de árvores de herança de classes. Também a distribuição espacial (*layout*) pode ser utilizada para expressar relacionamentos entre objetos que cumprem com uma dada propriedade, como por exemplo, classes que colaboram muito freqüentemente são desenhadas mais próximas que classes que colaboram pouco entre si. Os eventos visuais envolvem mudanças na apresentação gráfica, geralmente, em termos de animações. Eventos visuais capturam informação dinâmica que é transmitida ao usuário dinamicamente para sugerir o comportamento do programa.

No caso geral, as diferentes ferramentas combinam ambas técnicas para condensar maior informação na mesma apresentação. Em alguns casos estas representações são aumentadas com propriedades visuais como cor, ênfase que codificam diferentes aspectos não estruturais como grau de interação entre classes, freqüência de ativação, etc.

Nos sistemas de visualização de software orientado a objetos atuais podem distinguir-se duas formas básicas de apresentação visual da informação do programa: representações estruturais baseadas em grafos e representações sintetizadas que exploram atributos visuais como dimensionalidade e proximidade, para permitir inferir propriedades globais do comportamento do programa.

4.5.1.1 Representações estruturais

A maior parte das ferramentas de visualização de software orientado a objetos, existentes na literatura, utiliza alguma forma de representação diagramática. Por exemplo, representações estruturais de hierarquias de classes são comuns nos *browsers* gráficos providos por muitos dos ambientes de desenvolvimento de software orientado a objetos.

Representações estruturais, com correspondência com notações estabelecidas, são intuitivamente mais simples de associar, como demonstram as notações mais difundidas para análise e projeto orientado a objetos, mas a complexidade de grandes sistemas as torna também difíceis de manejar. A solução habitual para este problema é a utilização de múltiplas janelas com visões parciais ou alternativas de mesma informação, como por exemplo visões estáticas de hierarquias e relacionamentos utilizando diagramas de Rumbaugh, e visões dinâmicas de comportamento de instâncias utilizando grafos de interação. Esta solução, porém, também apresenta inconvenientes do ponto de vista da necessidade de integrar diferentes modelos visuais para compreender o funcionamento e a estrutura do programa. Do ponto de vista da compreensão de frameworks, a utilização das notações fornecidas pelos métodos de desenvolvimento sofre dos inconvenientes citados no capítulo anterior para a documentação de frameworks.

O *Program Explorer* é o principal exemplo da utilização de técnicas diagramáticas para a visualização de programas C++. A ferramenta fornece visualizações baseadas em grafos, para visualizar hierarquias de herança, interconexão de instâncias através de mensagens e grafos de interação de objetos [RUM 91]. Uma das aplicações mais relevantes da ferramenta para compreender frameworks é a visualização de padrões de projeto. Através da informação estática dos padrões de projeto que compõem um dado framework, expressada em termos de clausulas Prolog, a representação

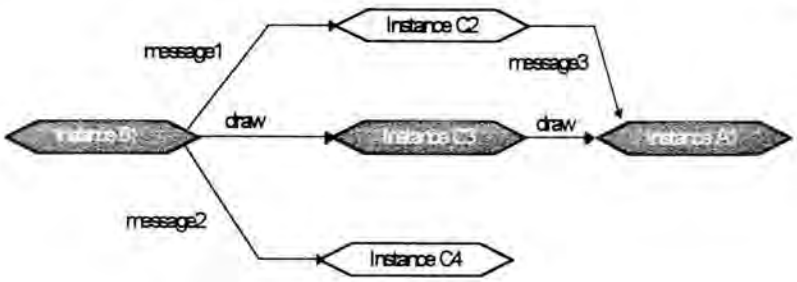


FIGURA 4.7- Representação visual de comunicação entre instâncias provida por *Program Explorer*

visual é utilizada para explorar um dado framework através da navegação entre as representações dos diferentes padrões que compõem o framework. Entretanto, as visualizações propostas se centram no comportamento das instâncias das classes que implementam ao padrão, não resultando simples determinar, sem informação adicional, qual o padrão visualizado nem como o padrão se materializa em termos da arquitetura da aplicação.

Object Visualizer fornece uma visualização diagramática dinâmica da forma em que os objetos comunicam-se, denominada *Agrupamento por Invocações Inter-Classes*, a qual explora atributos de distribuição espacial e cores para ressaltar visualmente o grau de interação entre classes. A visualização apresenta as classes distribuídas no espaço de acordo com a freqüência de comunicação entre elas. Quanto maior é a interação entre duas classes, aparecem desenhadas mais próximas entre si. Linhas azuis indicam o caminho de métodos executados em cada instante, enquanto uma linha vermelha indica cada método em execução em cada instante de tempo. Estaticamente, se algumas classes aparecem muito agrupadas então é uma sugestão de classes altamente acopladas ou um subsistema. Na visão dinâmica, atividade concentrada da linha vermelha, em torno de um conjunto de classes, indica um *hot-spot* de execução.

4.5.1.2 Representações sintetizadas

Representações sintetizadas são aquelas que utilizam técnicas que tentam prover mecanismos não diagramáticos para facilitar o reconhecimento visual de propriedades globais do comportamento do programa.

Object Visualizer é o exemplo mais significativo deste tipo de visualizações, através de múltiplas visões que apresentam informação resumida dos resultados de uma execução. As visões são representadas como matrizes cujas entradas são preenchidas com cores para visualizar a freqüência de criação e destruição de instâncias, invocações *inter* e *intra* classes, história de atribuição de instâncias, etc. Por exemplo, a ***Matriz de Instanciação*** (*Allocation Matrix*) permite visualizar quais classes produzem instâncias de outras classes. A representação bidimensional organiza o conjunto de classes em uma matriz, na qual cada entrada [Classe$_i$, Classe$_j$] contém o número de instâncias da Classe$_i$ foram criadas por instâncias da Classe$_j$. Estaticamente, a visão mostra o número relativo de instâncias criadas entre as diversas classes. Dinamicamente, mudanças na cor associada indicam quais classes criam instâncias e a taxa de criação segundo uma escala desde o vermelho (poucas) até violeta escuro (muitas). A ***Matriz de Chamadas Inter-Classes***, fornece uma visão acumulada das comunicações entre objetos, resumidas por classe. As classes são organizadas da mesma forma que na matriz de instanciação, aparecendo na ordem em que são instanciadas. Cada entrada [Classe$_i$, Classe$_j$] contém o número de métodos da Classe$_i$ invocados por métodos da Classe$_j$. A cor associada com cada entrada indica a taxa de invocações desde vermelho (poucas) até violeta escuro (muitas). Estaticamente, a ordenação de classes indica a sua ordem de instanciação. Linhas verticais acima da diagonal principal, podem indicar classes base muito utilizadas, seja por código herdado ou por invocações das suas subclasses. Entradas sobre a diagonal indicam invocações a *self*, enquanto que agrupamentos perto dela são uma sugestão de classes projetadas para trabalharem juntas. Regiões escuras, são também uma indicação de classes muito acopladas, devido ao fato de terem sido criadas juntas e possuírem uma grande troca de mensagens. Dinamicamente, o surgimento de muitas classes novas indica o começo de uma fase nova de execução, enquanto que mudanças na cor, centram a atenção em classes que interagem freqüentemente.

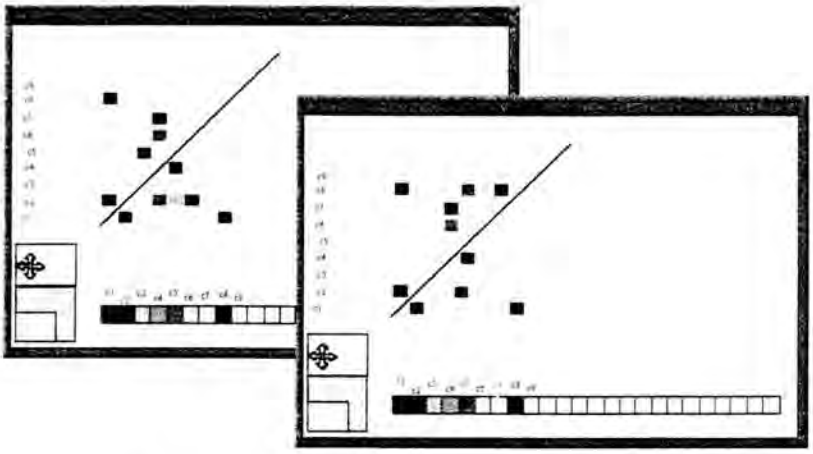


FIGURA 4.8- Representação de Matrizes utilizadas pelo Object Visualizer

A informação provida por estas visualizações é valiosa para compreender o funcionamento global de um programa, mas, do ponto de vista dos frameworks, não evidenciam com clareza os componentes abstratos, o fluxo de controle entre estes componentes, nem como eles são especializados.

Na área das visualizações estáticas, o *Browser de Afinidade* é uma das aplicações mais interessantes da utilização de propriedades visuais [PIN 90]. O *browser* explora o conceito de distância entre objetos para agrupar visualmente classes que referem-se a conceitos semelhantes. Para isto utiliza uma visualização baseada na tela dos radares (FIGURA 4.9), para representar a posição relativa de classes de acordo com algum critério de semelhança especificado. Através desta visualização é possível identificar rapidamente componentes de biblioteca para sua utilização de acordo com critérios de semelhança funcional.

As visualizações sintetizadas podem ser utilizadas para representar, por exemplo, o grau de interação de instâncias ou classes em grupos colaborativos, mas não oferecem grandes vantagens do ponto de vista da compreensão do comportamento detalhado de um framework. No entanto, podem servir como um complemento importante de visualizações estruturais que reflitam a estrutura estática e de controle do framework.

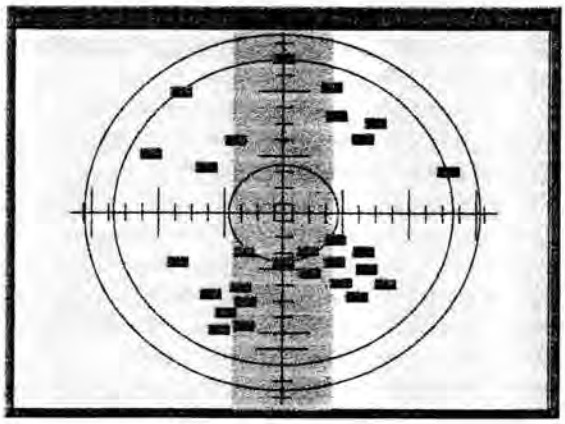


FIGURA 4.9- Visão de Radar

4.5.1.3 A terceira dimensão

Com o surgimento de hardware gráfico de baixo custo que suporta a síntese de gráficos tridimensionais em tempo real, novas interfaces e sistemas de visualização que aproveitam estas

capacidades estão atualmente sendo explorados. A incorporação de una nova dimensão na visualização apresenta múltiplas vantagens, como por exemplo, aumento na densidade de informação [ROB 91][ROB 93], legibilidade, representação de múltiplos diagramas bidimensionais [KOI 93], integração da informação local e global em uma só visualização [MAC 91] e captura da história de execução de uma visão bidimensional para detecção de padrões [BRO 93], entre outras.

Atualmente, existe uma tendência crescente na utilização de gráficos tridimensionais para a realização de apresentações. Existem já muitos sistemas específicos que utilizam esta tecnologia (sistemas de CAD, modelagem molecular, visualização científica, etc.). No entanto, o aspecto de sua utilidade efetiva para a visualização de software é ainda matéria de discussão e pesquisa.

As aplicações atuais de técnicas de visualização tridimensional para visualização de software, são uma generalização dos sistemas de visualização de programas, incorporando algumas noções de visualização científica para grandes volumes de dados. Estas técnicas atribuem uma semântica particular à terceira dimensão para interpretar dados de múltiplas apresentações bidimensionais [BRO 93] [KOI 93].

Na área de depuração de programas orientados a objetos, Vion-Dury e Santana [VIO 94] apresentam a utilização de visualização tridimensional baseada no conceito de imagens virtuais (*virtual images*), para a depuração de sistemas orientados a objetos distribuídos. Uma imagem virtual é uma representação de um objeto utilizando um modelo espacial 3D. Neste modelo os objetos são representados por poliedros que possuem formas, cores, volumes e orientações específicas. O conceito de *variável visual* é utilizado para associar os valores que o olho humano pode perceber e processar

TABELA 4.2- Semântica das variáveis visuais

Variável	Semântica atribuída
X,Y,Z	Informação Estrutural e Relacionamentos Globais entre objetos
Forma	Pertença a Classe
Cor	Atributos de Proteção
Tamanho	Tamanho de Memória
Orientação	Estados Internos

durante uma unidade de percepção com propriedades que identificam atributos particulares dos objetos observados (TABELA 4.2)

A ferramenta fornece diferentes visões estáticas 3D das hierarquias de classes e relacionamentos entre objetos distribuídos. Estas representações oferecem as vantagens típicas dos sistemas de visualização 3D devido as facilidades inerentes de navegação e múltiplas perspectivas. Entretanto, o aspecto mais destacado é a visualização de comportamento dinâmico de um sistema observado em tempo real através de visões 3D de propagação de mensagens como "espirais pulsantes" as quais relacionam os objetos que recebem e enviam mensagens no tempo. Este trabalho justifica muito bem as vantagens da utilização de gráficos 3D para a visualização de sistemas de objetos, mas não resulta evidente como a utilização de formas poliédricas pode substituir completamente o texto.

Técnicas básicas de visualização 3D são utilizadas também na área de visualização arquitetônica. *Mirror* [ORO 95] é um protótipo de ambiente de desenvolvimento de aplicações reflexivas C++, que utiliza visualizações tridimensionais simples baseadas em grafos, para visualizar arquiteturas reflexivas de meta-objetos. Os diferentes níveis da arquitetura são representados por planos normais ao eixo *y*. Os relacionamentos entre objetos de um mesmo nível são representados por linhas que unem os objetos relacionados, enquanto os relacionamentos entre objetos e meta-objetos são representados por linhas que unem os dois níveis. Cores são utilizadas para ressaltar a transferência de controle entre os diferentes níveis durante a execução do programa. (FIGURA 4.10).

A utilização de representações 3D é de utilidade, neste caso, para prover o usuário com um modelo semelhante ao utilizado mentalmente para imaginar a composição de uma arquitetura reflexiva. Esta similaridade ajuda muito para facilitar a compreensão geral da arquitetura. O mecanismo de navegação por translação da câmara permite o deslocamento do foco de atenção a

diferentes pontos da estrutura, bem como o mecanismo inerente de *zoom* permite visualizar tanto estrutura global como estrutura detalhada dos componentes.

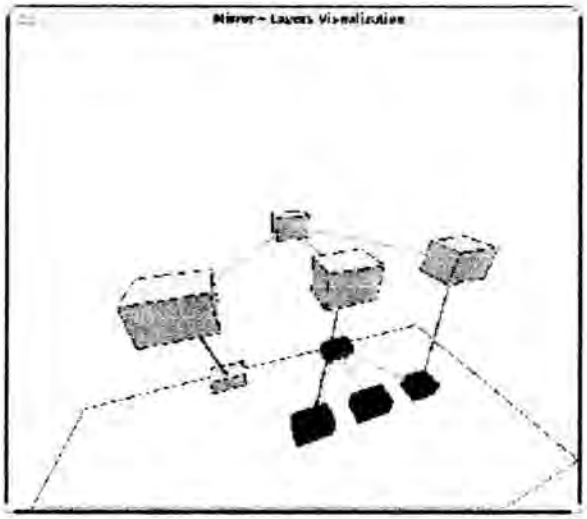


FIGURA 4.10- Visualização 3D de uma Arquitetura Reflexiva provida por *Mirror*

Este tipo de visualizações, entretanto, apresenta alguns inconvenientes, a saber:

- Para obter tempos de resposta adequados para seu uso, é necessário hardware especializado para graficação 3D, o qual não é facilmente disponível nas estações de trabalho atualmente utilizadas.
- A manipulação direta neste tipo de interfaces requer de dispositivos especializados que podem inibir a sua utilização conjunta com a manipulação textual do código.
- Representações 3D de conceitos que não possuam um mapeamento direto com as construções visualizadas podem interferir na facilidade de compreensão inicial da própria visualização, aumentando a curva de aprendizado da ferramenta.
- A implementação de visualizações tridimensionais é muito mais complexa que representações em duas dimensões, mesmo com as bibliotecas de suporte para graficação 3D hoje disponíveis.

Independente destas limitações, visualizações 3D poderiam servir como um complemento interessante de visualizações diagramáticas bidimensionais. Por exemplo, visualizações 3D podem ser de utilidade para representar em forma conjunta o fluxo de controle entre os diferentes níveis das hierarquias de classes que compõem o framework e as classes construídas pelo usuário. Uma representação desta natureza, ainda que prescindível, coloca facilmente em evidência a inversão de controle característica dos frameworks, tal como foi descrita na FIGURA 2.1.

4.5.1.4 Grafos de fluxo de mensagens

Da discussão das seções precedentes pode-se concluir que não é possível assegurar que um único tipo de visualização seja a solução para a compreensão de frameworks. Entretanto, representações estruturais ressaltando componentes abstratos, a categoria dos seus métodos (abstratos, *template*, *hook*, base) e como eles se relacionam através da troca de mensagens, podem prover uma primeira aproximação muito útil para compreender a organização global do framework. A visualização destes aspectos é particularmente útil para compreender quais componentes devem ser especializados e quais métodos devem ser implementados. Também é necessário visualizar, em algumas circunstâncias,

a hierarquia de especializações enfatizando os métodos que são acrescentados e a forma em que especializam o fluxo de controle do framework.

Tomando em consideração estes aspectos, uma visualização que auxilie no processo de compreensão de um framework deveria satisfazer os requisitos seguintes [CAM 95b]:

- *Visualização de Componentes Abstratos*: A visualização deve permitir identificar rapidamente os componentes essenciais da arquitetura, sua interface e a forma em que eles colaboram.
- *Visualização da hierarquia de especializações de componentes*: A hierarquia de especializações deve poder ser visualizada enfatizando os métodos que são acrescentados e a forma em que especializam o fluxo de controle do *framework*.
- *Visualização de Fluxo de Controle:* A maior parte do fluxo de controle previsto pelo *framework* é baseado nos métodos *template*, o qual pode ser estendido para cada aplicação especifica pelos outros métodos. Visualizar o *framework* é, em grande medida, visualizar estes possíveis caminhos de controle, evidenciando como os métodos estão relacionados através de troca de mensagens.
- *Rápida identificação de tipos de métodos*: Uma representação adequada para uma classe deve deixar evidente para o usuário quais métodos correspondem a cada categoria. A fácil identificação das categorias de métodos permite compreender quais métodos devem ser especializados e quais métodos fornecem a estrutura de controle de genérica.

A FIGURA 4.11 apresenta uma possivel forma de representar um componente de um framework segundo as considerações anteriores. Um componente será caracterizado por cinco partes : seus atributos (alguns dos quais podem ser referências a objetos componentes), uma parte genérica que define os métodos template; uma parte abstrata com os métodos abstratos; uma parte redefinivel que define os métodos hook, e, finalmente a parte que define o comportamento base, o qual não deveria ser redefinido.

Esta notação para os componentes, além de fornecer uma clara classificação dos métodos, serve também aos fins de especificar o fluxo de controle genérico *intra* e *inter* classes. O primeiro caso é constituido pelos métodos que invocam operações implementadas na própria classe ou em uma subclasse. O segundo caso corresponde às invocações a métodos implementados em outras classes.

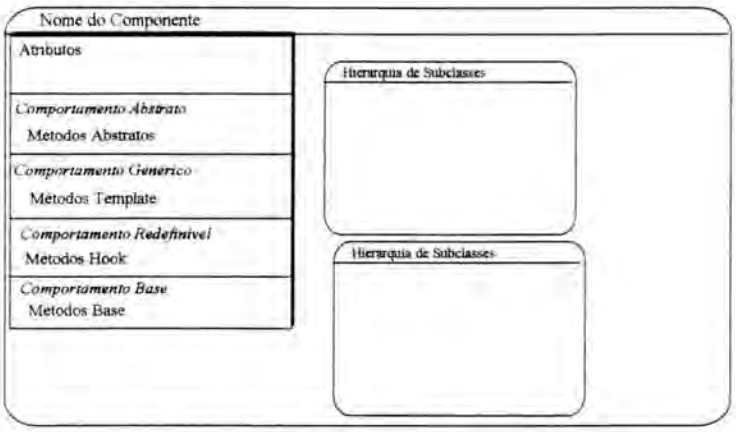


FIGURA 4.11- Representação de componente abstrato e hierarquias derivadas

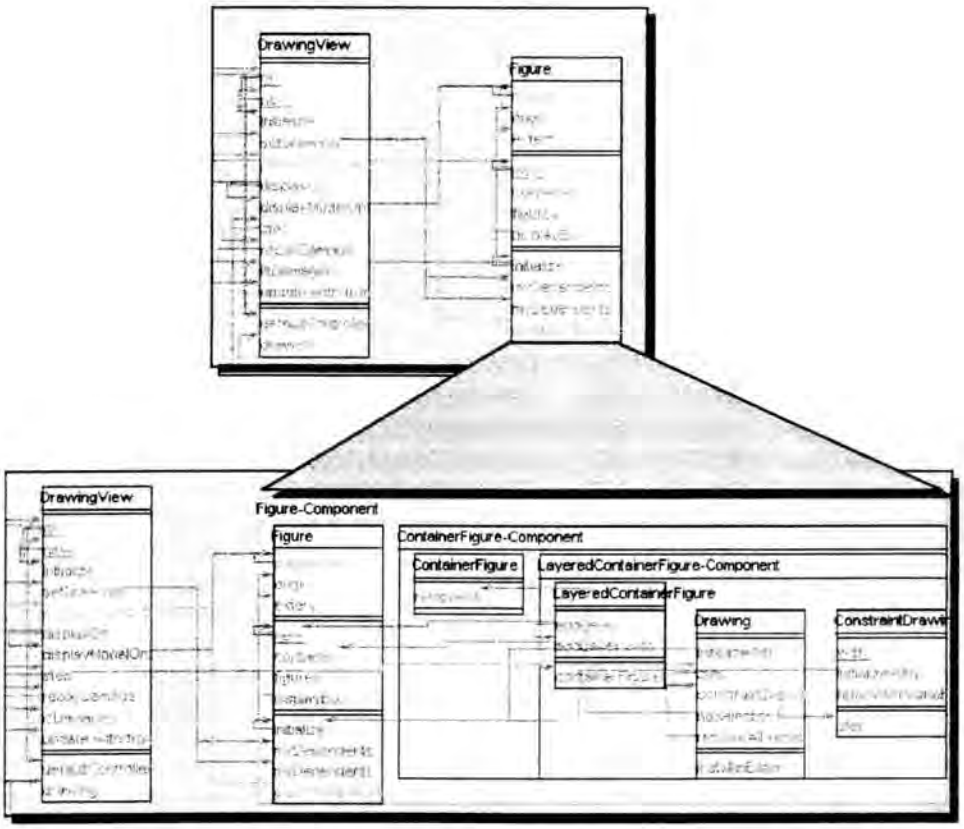


FIGURA 4.12- Visualização de componentes e hierarquias aninhadas

A FIGURA 4.12 ilustra a hierarquia de um componente abstrato Figure, na qual aparece o comportamento próprio da classe e um sub-componente *ContainerFigure*. Este, por sua vez, tem um sub-componente *LayeredContainerFigure* do qual herdam *Drawing* e *ConstraintDrawing*. Também é possível observar o fluxo de controle interno da hierarquia, o qual indica mensagens enviados a *self* ou *super* (em terminologia Smalltalk). O fluxo de mensagens *inter* e *intra* componentes é representado por setas que unem o método que envia a mensagem e o método ativado pela mensagem.

A informação das mensagens combinada com os tipos de métodos permite condensar, na apresentação do componente abstrato, uma grande quantidade de informação relativa a extensão de métodos *hook* e a forma em que instâncias são organizadas[1].

4.5.2 Exploração de informação: Navegação e Manipulação

A navegação é uma técnica que faz parte integral das atividades de compreensão. O processo de compreensão de um programa consiste, essencialmente, na exploração do espaço de informação que representa o programa analisado, através de apresentações visuais que facilitem a análise dessa informação desde diferentes perspectivas.

Todas as ferramentas descritas nas seções precedentes oferecem alguma capacidade para navegar entre as diferentes representações gráficas que fornecem. Estas capacidades são implementadas pela interface com o usuário e restritas a relacionamentos preestabelecidos pela ferramenta, o qual reduz as possibilidades de livre navegação entre informação relacionada.

No caso de uma representação de hiperdocumento, na medida que o tamanho da rede de informação cresce, o seguimento de vários elos ou nodos conduz freqüentemente ao problema de desorientação do usuário. A desorientação pode acontecer quando o usuário perde a noção do lugar na rede onde se encontra localizado, ou, pior ainda, quando perde o sentido de qual ou propósito que o

[1] Esta notação é descrita com mais detalhe no capítulo 7

levou a determinado lugar da rede [SHN 87]. Begeman e Conklin demostraram que mais de duas dúzias de nodos ou elos produzem problemas de percepção visual e espacial que rapidamente conduzem à desorientação do usuário [BEG 88]. Este efeito tem um papel importante para diminuir a efetividade da navegação como técnica de auxilio para a compreensão de informação complexa, a menos que mecanismos adicionais sejam providos para permitir ao usuário a conceitualização do espaço de informação com um todo.

Representações de hiperdocumento que permitam reduzir a complexidade da rede, como o modelo de contextos antes citado, e capacidades de consulta que permitam acessar diretamente informação de acordo com propriedades ou relacionamentos especificos, são dois elementos importantes que contribuem para aliviar os problemas gerados pela desorientação. Também a utilização de interfaces de manipulação direta, é um outro elemento de importância fundamental para facilitar a exploração da informação do programa desde representações visuais até o código.

Interfaces de manipulação direta permitem ao usuário interagir com um documento expressando intenções através de dispositivos como o *mouse* ou o *joystick*. Este tipo de interfaces habilita técnicas de exploração, além do mecanismo básico provido pelas barras de deslocamento (*scrolling*), que favorecem a análise detalhada de visualizações diagramáticas, como por exemplo, seleção de porções de informação, o deslocamento do documento (*paning*) e o *zooming*.

Estas funcionalidades são importantes para o caso de representações diagramáticas de grandes sistemas. A limitação de espaço físico imposto pela tela, obriga ao usuário a visualizar parcialmente os diagramas, dificultando a compreensão da estrutura completa do sistema. Para isto, o *zoom contínuo* [MAR 95] e o *zoom semântico* [MUT 95] são duas técnicas que estão sendo atualmente exploradas para ajudar a diminuir a desorientação do usuário produzida pelas múltiplas janelas habitualmente geradas pelos sistemas navegacionais convencionais.

Através de zoom contínuo uma visualização pode ser gradualmente aumentada ou reduzida sob controle direto do usuário, permitindo visualizar tanto o documento completo como partes específicas dele.

O *zoom* semântico permite que a apresentação dos elementos visualizados seja dependente do nível de zooming corrente. Estas apresentações podem consistir de apresentações fixas de um mesmo elemento ou de procedimentos que geram a apresentação em função do nível de detalhe. Este mecanismo foi introduzido pelo sistema *Galaxy of News*, no qual o *zooming* é equivalente com uma operação de filtragem interativa. Quando o usuário se aproxima a um item de informação, a apresentação se reestrutura para mostrar toda a informação relativa a esse item. Desta forma, alguns elementos presentes originalmente na apresentação desaparecem da mesma e surgem outros não visíveis.

Neste trabalho, um conceito semelhante ao zoom semântico é utilizado para evitar a geração de múltiplas janelas no caso de visualizações que admitem diferentes graus de detalhe ou abstração, através do mecanismo de abstratores descrito mais abaixo.

4.6 Análise e Filtragem de Informação

A facilidade para derivar e representar abstrações da informação coletada é um aspecto central nos sistemas de engenharia reversa e de visualização de software. O termo abstração pode admitir um significado dual, derivado do processo através do qual a abstração é obtida:

- abstração como resultado da análise da informação do programa através da qual relacionamentos funcionais implícitos no código são deduzidos, como por exemplo subsistemas.
- abstração como resultado de um processo de filtragem de informação não relevante, para gerar representações sintéticas da informação.

4.6.1 Recuperação de abstrações

Neste sentido, uma abstração denota aquelas construções que não são habitualmente suportadas em forma explícita pela linguagem de programação, mas que formam parte integral da forma na que o software é conceitualizado pelo projetista, como por exemplo subsistemas. Isto é particularmente válido para sistemas orientados a objetos e frameworks. Subsistemas [WIR 90], grupos colaborativos [HEL 90] ou padrões de projeto [GAM 94] representam abstrações de projeto que não são suportadas pelas linguagens orientadas a objetos atuais, mas que são de grande importância para compreender como os objetos de um sistema são organizados e colaboram para satisfazer a funcionalidade global.

Este tipo de abstrações podem ser recuperadas, ao menos parcialmente, através da análise da informação coletada. Para esta atividade, o *pattern matching* é uma das técnicas mais importantes. Grande parte do processo de compreensão consiste, essencialmente, no reconhecimento de padrões estruturais conhecidos, seja mentalmente pelo programador ou automaticamente pela ferramenta. Técnicas de *pattern matching* são utilizadas pela maioria das ferramentas de engenharia reversa, mas no caso geral, são limitadas a problemas de pequena escala. Representações baseadas em grafos apresentam o problema da impossibilidade computacional de verificar a existência de um dado padrão misturado entre um conjunto de padrões intercalados no código de um programa [FIS 91]. No caso de programas orientados a objetos, o reconhecimento de padrões estruturais através de *pattern matching* apresenta limitações semelhantes, aprofundadas pela natureza dinâmica do grafo definido pela configuração de instâncias. Representações baseadas em Prolog podem simplificar o processo de casamento, baseando-se em propriedades abstratas do fluxo de controle, mais que na estrutura do mesmo [LAN 95].

A identificação de padrões de projeto [GAM 94] pode ser de muita utilidade para compreender como determinadas partes do framework estão organizadas e colaboram entre si [LAN 95]. Entretanto, uma abordagem centrada exclusivamente no reconhecimento e visualização deste tipo de padrões, não resulta totalmente adequada para orientar efetivamente o processo de compreensão de um framework. Nem todas as estruturas de um framework correspondem a este tipo de padrões e o número de padrões atualmente reconhecido é relativamente baixo, variando muito no nível de abstração e domínio de aplicação. Além disto, a materialização de padrões de projeto depende fortemente da linguagem de implementação e dos requisitos particulares de cada aplicação, o que implica na impossibilidade prática de garantir que todos os possíveis padrões que formam o projeto de um framework possam ser automaticamente reconhecidos.

Um padrão expressa, essencialmente, uma intenção de projeto, descrevendo uma estrutura genérica de classes para organizar e distribuir responsabilidades entre elas, a qual pode ser idêntica em muitos casos. Por exemplo:

- *Composite* e *Decorator* possuem estruturas semelhantes, mas seus objetivos são completamente diferentes. Enquanto o *Decorator* acrescenta funcionalidades ao objeto que decora, o Composite tem por objetivo a composição de objetos. Do ponto de vista do comportamento dinâmico, ambos padrões caracterizam-se por propagar mensagens aos seus componentes. A única diferença essencial é que o *Decorator* está composto por só um componente, enquanto o *Composite* tem, habitualmente, mais de um.

- *Decorator* e *Proxy* possuem uma estrutura de classes idêntica, diferençando-se na intenção de projeto. O objetivo de um Proxy é controlar o acesso ao objeto que representa. Assim, o comportamento diferençado entre ambos padrões é que, no caso geral, um *Decorator* propaga sempre as requisições para seus componente, enquanto o *Proxy* pode propagar condicionalmente as requisições.

- *State* e *Strategy* também possuem estruturas idênticas. O *Strategy* encapsula um algoritmo como um objeto, enquanto o *State* encapsula um estado como um objeto,

o qual é acessado pela sua interface. A diferença entre ambos padrões reside em que o *Strategy*, geralmente encapsula uma única requisição, definido um único método abstrato, o qual define a interface, que será redefinida pelas estratégias concretas. No *State*, por sua vez, cada subclasse que representa um estado, freqüentemente, implementará um conjunto de métodos de acesso a esse estado. Portanto, é provável que a classe *State* defina mais de um método abstrato.

A identificação de subsistemas e sua representação visual também pode fornecer uma aproximação inicial da arquitetura da aplicação. Freqüentemente, funções de um framework são divididas em conjuntos de classes que formam um grupo colaborativo. Através da análise do fluxo de controle é possível aproximar a constituição de tais grupos. Mas, neste caso, também não é possivel garantir que o agrupamento responde a um critério funcional exato, sem conhecimento da semântica das colaborações. Assim é necessário que uma ferramenta forneça os meios para permitir ao usuário decidir quando um agrupamento ou padrão sugerido é adequado ou deve ser modificado ou refinado.

4.6.2 Filtragem de informação

Neste contexto o termo abstração é utilizado para denotar um nível de detalhe determinado de descrições ou visualizações, que só inclui aspectos relevantes da informação coletada, de acordo com alguma semântica fixada pela ferramenta ou selecionada pelo usuário.

Representações desta natureza, são, habitualmente, o resultado de filtrar a informação do programa de acordo com propriedades das entidades envolvidas ou de consultas por relacionamentos e propriedades selecionadas pelo usuário. A filtragem de informação segundo critérios estabelecidos pode ser um complemento para auxiliar ao usuário na procura de abstrações que não podem ser automaticamente derivadas.

Dentre as diversas alternativas de filtragem de informação existentes, o fatiado (*slicing*) [WEI 84] é a técnica predominante entre as ferramentas de engenharia reversa. O fatiado consiste na divisão do código de um programa em fragmentos que podem afetar o valor de uma ou várias variáveis. Isolando essas sentenças, se reduz, consideravelmente, a carga cognitiva necessária para compreender uma porção de código potencialmente muito grande. Cada fatia pode ser apresentada de forma textual ou gráfica.

O conceito de fatiado pode também ser aplicado para explorar a estrutura de controle de um framework, se considera-se, por exemplo, a seleção daqueles fluxos de controle que envolvem aspectos relevantes da estrutura de controle genérica. A representação fornecida pelo *Program Explorer* habilita, por exemplo, a definição de regras de seleção que permitem visualizar as instâncias relacionadas por troca de mensagens que cumprem com alguma propriedade determinada pelo usuário. A aplicação de diferentes regras produzirá uma visão particular das colaborações entre as instâncias selecionadas.

Tomando em consideração as características estruturais dos frameworks, discutidas no capítulo 2, construções características podem ser identificadas através do contexto no qual as mensagens são enviadas e recebidas. Por exemplo, caminhos de ativação de métodos abstratos, hook e *template*, ou caminhos de controle que conduzem à criação de instâncias, fornecem muita informação para compreender como funções específicas do framework foram projetadas, quais os pontos de especialização previstos e como os diferentes componentes do framework são interconectados no momento da sua criação.

Por exemplo, fluxos que ativam métodos hook ou abstratos são úteis para filtrar a visualização de acordo a uma perspectiva de meta-padrões [PRE 94]. Neste caso, métodos template e métodos hook relacionados serão selecionados, permitindo visualizar os pontos de extensão do framework. A partir de uma perspectiva de padrões de projeto, a análise de fluxos recursivos, isto é, mensagens que são propagadas entre instâncias pertencentes a classes da mesma hierarquia, permite ao usuário buscar pela existência de alguma materialização dos padrões compostos, como *Composite*, *Decorator* ou *Proxy*. Fluxos propagativos entre múltiplas instâncias podem também sugerir a existência

de um padrão *Chain of Responsibility*, isto é, uma seqüência da mesma mensagem enviada a diferentes instâncias com diferente freqüência. Uma análise detalhada do código e das instâncias envolvidas pode ajudar a determinar se o comportamento codificado pelas classes corresponde com alguns destes padrões. A análise do fluxo criacional pode ser de utilidade para reconhecer padrões criacionais como por exemplo o *Abstract Factory*.

4.6.3 Aspectos de arquitetura e adaptabilidade das ferramentas

A construção de visualizações, e particularmente capacidades de manipulação direta, é uma das tarefas que mais custo de implementação implica na construção de ferramentas de visualização[2]. A maior parte das ferramentas discutidas nas seções precedentes, fornecem pouca ou nenhuma facilidade para a construção de visualizações alternativas, limitando, em conseqüência, sua adaptabilidade. Na maioria dos casos a construção de visualizações é suportada por algum framework de interfaces (por exemplo *Object Visualizer* e o *Program Explorer* baseiam suas interfaces com o usuário no framework *Interviews* [VLI 89]). Estes frameworks implementam alguma variante do modelo MVC descrito no capítulo 2, o qual induz uma arquitetura que atribui ao componente que implementa a visão a responsabilidade de produzir a visualização. Com esta estrutura, diferentes visões da informação representada pelo modelo devem ser programadas *ad-hoc*, requerendo que cada tipo de visualização recupere a informação necessária e produza a representação gráfica de acordo com uma notação determinada (FIGURA 4.13).

Esta abordagem, embora garantindo a independência da representação a respeito das suas visões, apresenta duas limitações essenciais, a saber:

- **Análise de abstrações**: A estrutura MVC não favorece a implementação de mecanismos de derivação de abstrações e filtragem de informação em forma independente da representação da informação do programa analisado ou das visualizações. Sob uma estrutura baseada em um framework derivado do modelo MVC, os algoritmos de análise da informação devem ser implementados como parte da representação ou como parte da visualização. Em ambos casos a solução torna-se dependente de aplicações particulares e, fundamentalmente, requer a extensão de classes existentes com comportamento que não depende do conceito que elas representam, mas das necessidades de localizar esse comportamento em alguma classe dentro do projeto. Isto é, estruturas deste tipo podem induzir projetos difíceis de extender e adaptar.
- **Filtragem contínua de níveis de abstração**: No caso geral, uma abstração é definida em termos de construções de mais baixo nível, como por exemplo, subsistemas são definidos como um conjunto de classes, ou hierarquias de classes, e suas colaborações. Uma classe, por sua vez, pode ser visualizada desde o ponto de vista da interface externa constituída pela assinatura dos métodos públicos, desde o ponto de vista da sua estrutura interna de variáveis, ou do código dos métodos. Idealmente, cada um destes aspectos deveriam poder ser visualizados ou ocultados de acordo ao nível de detalhe desejado pelo usuário. No entanto, com uma abordagem baseada exclusivamente no modelo MVC, os mecanismos para filtrar os diferentes níveis de detalhe, ou abstração, devem ser implementados para cada visualização particular, tornando muito complicada a programação de novas visualizações e reduzindo, também, a sua reutilizabilidade.

[2] A complexidade da utilização de um framework de interfaces, como MVC, foi discutida e exemplificada no capítulo anterior. Embora um framework de interfaces possa oferecer muitas vantagens para a construção de interfaces de diálogos, a construção de interfaces de manipulação direta e visualizações complexas, é uma tarefa que requer esforço muito considerável.

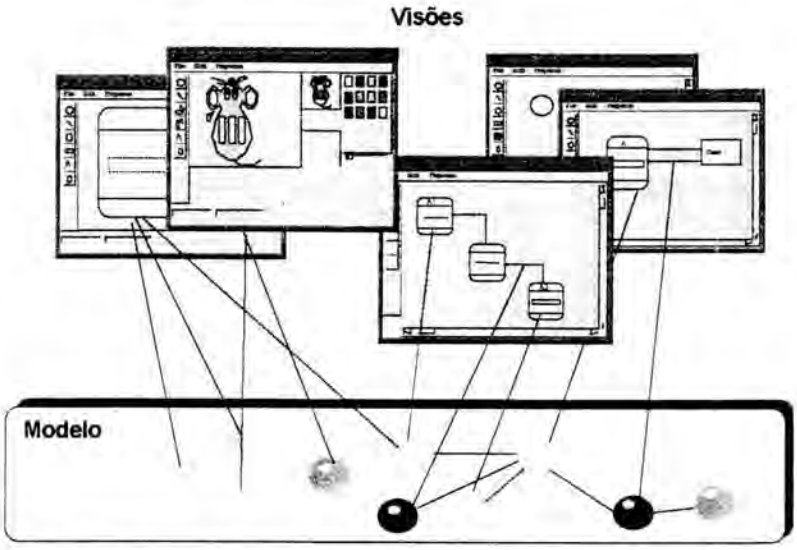


FIGURA 4.13- Correspondência entre visões e modelo em MVC

Estas limitações, obviamente, podem ser resolvidas de várias formas diferentes. Neste trabalho, o conceito de ***abstrator*** foi desenvolvido para integrar, por composição de objetos, algoritmos de análise de abstrações com mecanismos de *zoom* semântico, que evitam a programação de mecanismos particulares de ocultamento de niveis de abstração.

4.6.3.1 Abstratores

Um abstrator é uma estrutura de projeto orientado a objetos que introduz um nível intermediário entre os objetos visuais e a representação de informação (FIGURA 4.14). Operacionalmente, um abstrator pode agir como *filtro*, *seletor* ou *gerador* da informação que uma visualização mostrará graficamente, encapsulando o comportamento necessário para produzir a informação que será visualizada.

Segundo o mecanismo convencional de comunicação entre visões e o modelo em MVC, os objetos que representam as visões solicitam a informação a ser mostrada através de mensagens enviados a seu modelo. Um abstrator substitui o objeto que representa o modelo e decide quando solicitar a informação requisitada pela visão ao modelo original. Deste modo o abstrator pode se comportar de três formas diferentes:

- De ponto de vista de geração de informação, um abstrator pode encapsular o algoritmo para reconhecer subsistemas ou grupos colaborativos e prover cada subsistema reconhecido como um dado a ser visualizado.
- Do ponto de vista de seleção de informação um abstrator pode ser projetado para selecionar, de acordo com algum critério estabelecido interna ou externamente, qual a informação a ser visualizada. Os seletores permitem especificar um critério de seleção da informação a ser visualizada, como por exemplo, visualizar só aquelas classes que estão relacionadas através de mensagens que ativam métodos definidos como abstratos em alguma superclasse. Estes critérios de seleção podem ser manipulados interativamente pelo usuário, fornecendo desta forma a possibilidade que uma mesma apresentação seja variada no seu nível de detalhe de acordo com o tipo de informação desejada pelo usuário.
- Do ponto de vista de filtragem, um abstrator pode decidir quando um determinado dado será visivel ou não, simplesmente fazendo-o disponivel para a visualização. Deste modo, através do uso de abstratores, as visualizações só devem se preocupar

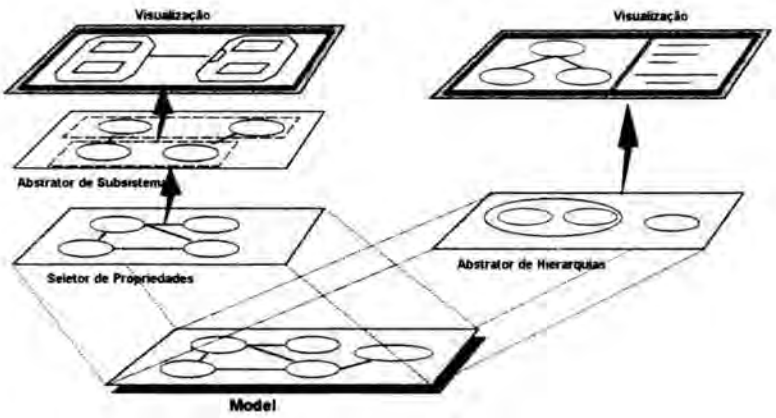


FIGURA 4.14- Composição de Abstratores

com mostrar graficamente a informação disponivel, sem tomar em consideração o grau de detalhe das mesmas.

O grau de detalhe da informação apresentada pode ser controlado pelos abstratores, permitindo implementar mecanismos de transformação gradual do nível de detalhe à maneira de um zoom contínuo da visualização.

Este mecanismo de zoom pode ser controlado através da definição de *escalas de abstração*. Uma *escala de abstração* define uma seqüência ordenada de conceitos que representam diferentes níveis de abstração de uma hierarquia. Por exemplo, a escala abaixo pode ser utilizada para definir o grau de detalhe de uma visualização baseada em subsistemas, como por exemplo, o grafo de colaborações [WIR 90]:

(subsystemAbstracion abstractHierarchy concreteHierarchy abstractMethod concreteMethod)

A seleção de algum nível nesta escala definirá que tipo de informação a visualização receberá para ser apresentada. Isto é, se o nível de abstração selecionado é *abstractHierarchy* a visualização só receberá a informação de quais os subsistemas e as classes do topo das hierarquias que compõem cada subsistema. Após, se o nível de abstração selecionado é *abstractMethod*, a visualização receberá os subsistemas, as hierarquias completas de classes e os métodos abstratos definidos nessas classes.

Deste o modo o programador de uma visualização composta só deve preocupar-se com a programação dos aspectos de apresentação visual da informação que o abstrator lhe fornece, evitando a necessidade de programar complexos mecanismos gráficos para o ocultamento e redimensionamento de visualizações.

4.7 Conclusão

Da discussão efetuada nas seções precedentes surge que a capacidade de adaptação das ferramentas é um dos aspectos principais a serem contemplados para adequarem-se a diferentes capacidades cognitivas, nível de conhecimento e preferências dos usuários. Especialmente, não é possivel assegurar que um estilo único de visualização seja a solução para a compreensão de frameworks. Isto impõe a necessidade de fornecer mecanismos que permitam adaptar uma ferramenta para cada uma das atividades envolvidas no processo de compreensão.

Dos mecanismos citados destacam-se os seguintes pontos:

- Através de visualizações abstratas da arquitetura de um framework é possivel prover uma estrutura inicial que guie a um usuário na exploração do framework

através de múltiplas visões adicionais, tanto no nível da organização de classes quanto no nível de comportamento das instâncias.

- A utilização de técnicas de reflexão computacional baseadas em meta-objetos, adaptadas para linguagens não reflexivas, aparece como uma solução conveniente para favorecer a adaptabilidade dinâmica dos mecanismos de captura de informação do programa analisado, e suportar a estratégia orientada pela visualização da arquitetura citada acima.
- A organização desta informação em termos de um modelo de hipertexto avançado, favorece o suporte de livros de projeto e a navegação entre diferentes visualizações que podem apresentar aspectos parciais da estrutura estática e dinâmica da aplicação analisada e, em conseqüência, do framework que a suporta.
- A navegação pode ser facilitada através de interfaces de manipulação direta que habilitem a seleção e o *zooming* das apresentações visuais, fornecendo mecanismos alternativos de exploração.
- O mecanismo de abstratores favorece a separação das funcionalidades de análise e filtragem de informação, permitindo a implementação de visualizações variáveis de acordo com escalas de abstração externamente definíveis.

Estes mecanismos, organizados na forma de um *framework* provem um suporte adaptável para a construção de ferramentas adequadas às diferentes necessidades e preferências dos potenciais usuários. Este framework, denominado Luthier, é descrito no capitulo 6.

5 Reflexão na Orientação a Objetos

No capítulo anterior apresentou-se sinteticamente o conceito de reflexão computacional, colocando a ênfase na sua variante de meta-objetos, como o mecanismo alternativo de captura de informação sobre o qual se sustenta este trabalho. Este capítulo tem por objetivo aprofundar nos conceitos de reflexão computacional nas linguagens orientadas a objetos em suas diferentes variantes, e analisar as alternativas para dotar a linguagens não reflexivas com o suporte necessário para implementar mecanismos reflexivos baseados em meta-objetos. Com base nesta análise o conceito de *gerenciadores de meta-objetos*, sobre o qual fundamenta-se a abordagem reflexiva adotada neste trabalho, é apresentado.

5.1 Modelos de Reflexão

Um sistema computacional é denominado *reflexivo* quando é capaz de realizar computação sobre sua própria computação [MAE 87], ou seja, é capaz de analisar (refletir sobre) e modificar o seu próprio comportamento. A capacidade de reflexão pode ser dividida em dois aspectos principais [GAB 91]: *introspeção* e *efetuação*. Introspeção é a capacidade de um programa de inspecionar as entidades que definem sua computação e, portanto, raciocinar acerca de seu próprio estado. Efetuação é a capacidade para alterar esse estado.

A reflexão computacional não é um conceito recente. Tanto linguagens funcionais clássicas como Lisp quanto linguagens de programação em lógica, como por exemplo Prolog, fornecem diferentes capacidades para implementar computações reflexivas. Entretanto, a capacidade de reflexão tornou-se um conceito de importância, e aplicabilidade prática crescente, com o advento das linguagens orientadas a objetos. Reflexão computacional no modelo de orientação a objetos, refere-se à obtenção de dados sobre as propriedades e comportamento de classes e de objetos de uma aplicação, e a possibilidade de modificar esses dados, modificando-se, em conseqüência, as propriedades das classes e dos objetos da aplicação.

Linguagens reflexivas orientadas a objetos se caraterizam por uma arquitetura em dois níveis [MAE 87]:

- ***Nível base***: contém os objetos do programa que realizam computações relativas ao domínio de aplicação ou, segundo Maes, o domínio externo do sistema.
- ***Nível meta***: composto de objetos que realizam computações sobre o domínio do sistema computacional, materializado pelos objetos do nível base. O domínio computacional, ou domínio interno do sistema, refere-se a todas as informações relativas às estruturas e mecanismos que habilitam a execução do programa. Uma alteração do domínio interno realizada pelo meta-nível fará com que o comportamento do nível base mude em conseqüência. Esta característica é denominada por Maes como *conectividade causal*.

Ferber identificou dois modelos básicos de reflexão na orientação a objetos [FER 89] denominados de *reflexão estrutural* [COI 87] e *reflexão comportamental* [MAE 87] (FIGURA 5.1). No primeiro caso, o meta-nível é composto por *meta-classes*, as quais contém informações sobre os aspectos estruturais do nível base, como descrição de variáveis e de métodos que irão compor todas as suas instâncias. Se esta estrutura pode ser alterada então a estrutura e comportamento de todas as instâncias também serão alterados. No segundo caso, o meta-nível é composto por *meta-objetos*, os quais contém informações sobre os aspectos de comportamento dos objetos do nível base, como por exemplo, como um determinado objeto trata uma mensagem. As seções seguintes descrevem mais detalhadamente estes conceitos.

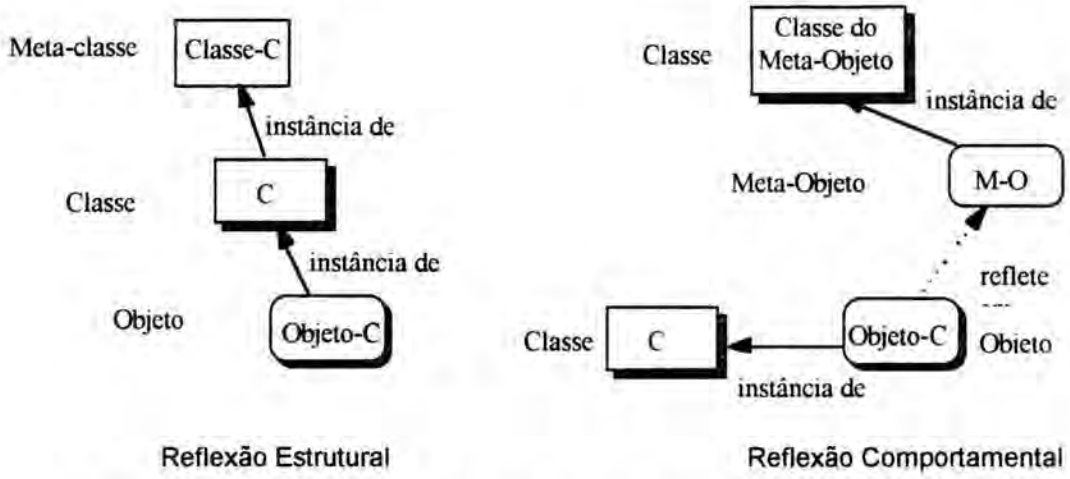


FIGURA 5.1- Relacionamento entre classes, meta-classes, objetos e meta-objetos

5.1.1 Reflexão Estrutural: O modelo de meta-classes

Algumas linguagens orientadas a objetos utilizam meta-classes para descrever a estrutura de todas as suas classes. Uma meta-classe é uma classe que descreve a estrutura de uma outra classe. Dependendo da linguagem de programação, classes e meta-classes podem ser representadas por objetos. Isto permite que as meta-classes possuam capacidades comportamentais para a instanciação dinâmica de classes, da mesma forma que as classes permitem a instanciação de objetos. Esta propriedade permite que meta-classes possuam a capacidade de realizar computações sobre seus próprios dados, fornecendo serviços a seus clientes através de métodos que permitem alterar aspectos estruturais da classe como por exemplo métodos, variáveis de classe e herança.

Existem, basicamente, duas possibilidades de estruturação de linguagens com meta-classes [LIS 95]:

- *Classes são instâncias de uma única meta-classe.* Todos os objetos são instâncias de uma classe e todas as classes são instâncias de uma classe ancestral comum, denominada de meta-classe. A meta-classe fornece uma interface para a estrutura de qualquer classe. Isto permite que as classes sejam consideradas como objetos do programa acessíveis ao programador. O exemplo típico desta abordagem é a linguagem Smalltalk, a qual associa uma instância da classe *MetaClass* com cada classe criada no ambiente.

- *Todas as classes possuem meta-classes.* Uma classe sempre possui uma meta-classe e todas as meta-classes possuem uma classe ancestral comum. Portanto, existem diversos níveis de classes: a classe de um objeto; a meta-classe da classe do objeto; a meta-classe, da qual foi especializada a meta-classe; a meta-classe; a classe da meta-classe. Exemplos clássicos desta abordagem são as linguagens orientadas a objetos baseadas em Lisp como LOOPS [STE 83], e o atual padrão CLOS [GAB 91].

Meta-classes, quando acessíveis, permitem a realização de *reflexão estrutural*, ou seja, toda a atividade realizada por uma meta-classe $\mathbf{M_X}$, com o objetivo de obter informações e realizar transformações sobre a estrutura estática de uma classe **x**.

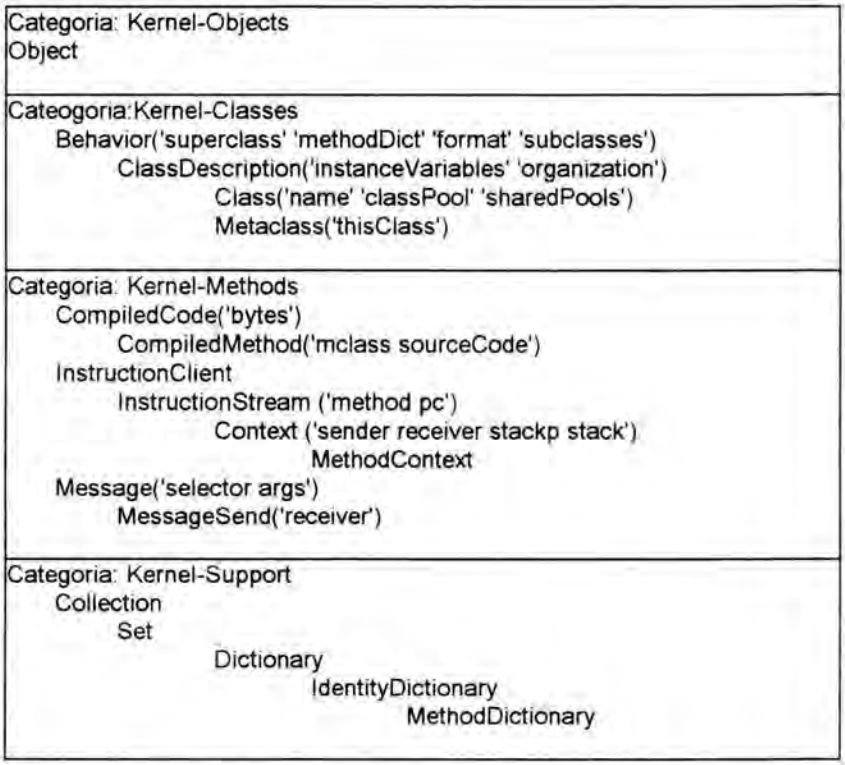


FIGURA 5.2- Hierarquia das principais classes do sistema Smalltalk -80

Smalltalk provê o exemplo mais difundido de capacidades de reflexão estrutural. Tanto o ambiente como o compilador são implementados através de classes e instancias de essas classes, constituindo a auto-representação do sistema Smalltalk-80. As principais classes que implementam o suporte de reflexão são mostradas na FIGURA 5.2, agrupadas conforme à categoria a qual pertencem.

Object é a superclasse de todas as classes, a qual fornece o comportamento comum para qualquer objeto, tal como imprimir ou copiar objetos e métodos de acesso à classe do objeto. A classe *Behavior* é a raiz da hierarquia de classes que fornecem o comportamento básico que habilita a reflexão estrutural nos aspectos seguintes:

- obter informações sobre a classe **x**: superclasses, subclasses, instâncias, métodos e variáveis
- alterar a estrutura da classe **x**: acrescentar ou eliminar variáveis e métodos
- atuar sobre a hierarquia de classes: criar novas classes, eliminar classes existentes e renomear classes.

A subclasse abstrata, *ClassDescription*, acrescenta algumas facilidades descritivas, como variáveis de instância, organização de categorias para os métodos, a noção de nome para a classe (que deve ser implementada pelas subclasses), a manutenção do registro de alterações (*Change set*) e a maior parte dos mecanismos para geração textual da descrição (*fileOut*)

Instâncias da classe *Class*, ou seja classes, descrevem a estrutura e o comportamento dos objetos. Sendo as classes também objetos, o seu comportamento é descrito por uma meta-classe, a qual é instância da classe *Metaclass*. Este comportamento, inclui mensagens para a inicialização de variáveis de classe e mensagens de criação de instâncias especificas para essa classe (*new* e *basicNew*).

No caso geral, a hierarquia de meta-classes é paralela à hierarquia de classes. Existe, entretanto, uma particularidade com a classe *Object*. Neste caso, a hierarquia de classes termina, mas a hierarquia de meta-classes deve continuar retornando a *Class* (note-se que todas as meta-classes são subclasses de *Class*).

Os métodos que implementam o comportamento dos objetos de uma classe estão relacionados em um dicionário (*methodDict*) definido na classe *Behavior*. Esse dicionário é uma instância da classe *MethodDictionary*, a qual fornece funções especializadas para agregar, eliminar ou recuperar elementos. Cada elemento do dicionário é uma instância da classe *CompiledMethod*, sendo identificado pelo seletor do método, e mantém uma referência ao código fonte (*sourceCode*) que descreve o método compilado (*bytes* em *CompiledCode*) e uma outra referência à classe (*mclass*) a qual pertence (FIGURA 5.3).

O comportamento definido por uma classe pode ser modificado dinamicamente através dos métodos que permitem acessar o dicionário de métodos, para agregar ou modificar métodos (*addSelector:withMethod:*), recompilá-los (*compileAll*, *recompile:*), etc. Por exemplo, a mensagem mostrada abaixo agregará um novo método chamado *aSelector* (ou modificará sua implementação se o método já existe) ao dicionário da classe *aClass*:

```
aClass addSelector: aSelector withMethod: aCompiledMethod
```

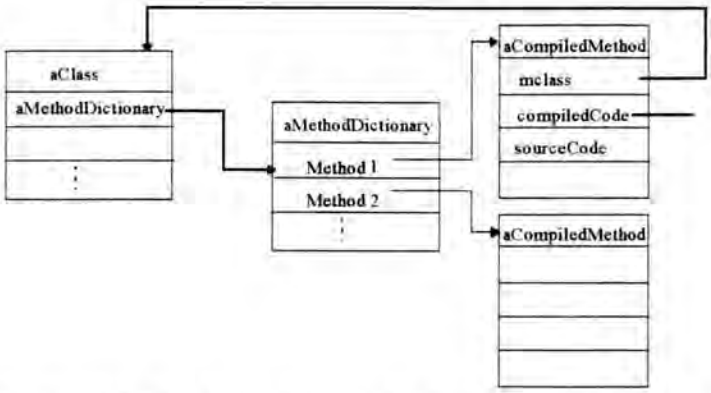


FIGURA 5.3- Configuração parcial de um dicionário de métodos

5.1.2 Reflexão Comportamental: O modelo de meta-objetos

Na reflexão comportamental de objetos, o *comportamento computacional* de cada objeto que compõe uma aplicação é determinado por um meta-objeto. Basicamente, o comportamento dos objetos no nível base é determinado pelos meta-objetos através do controle da ativação dos métodos (FIGURA 5.4). Assim, novos comportamentos podem ser acrescidos através da atuação sobre o processo de instanciação ou modificando a forma como os métodos são executados. Um meta-objeto, por exemplo, pode delegar a execução de um dado método a um outro objeto, ou ainda, distribuir esta responsabilidade entre um conjunto de objetos.

O modelo de meta-objetos se diferencia do modelo de meta-classes, principalmente, por sua associação por objetos e não por classes. Qualquer objeto da aplicação pode ser associado a um ou mais meta-objetos, que definem, implementam ou participam de diferentes formas, da execução dos objetos do nível base. Suas principais características são [LIS 95]:

- *Individualidade*: a cada objeto de nível base pode ser associado um meta-objeto. Do ponto de vista de estruturação de aplicações, isto significa que podem ser selecionadas determinadas instâncias de classes para a realização de reflexão computacional, permanecendo não reflexivos outros objetos instanciados a partir das mesmas classes.
- *Separação de Classes*: a classe do meta-objeto é distinta da classe do objeto associado, permitindo que objetos de uma mesma classe sejam associados a

diferentes meta-objetos; inversamente, meta-objetos instanciados a partir de uma mesma classe podem ser associados a objetos de nível base de diferentes classes.

- *Reflexão comportamental*: a classe de um meta-objeto contém informações sobre seu referente, incluindo suas variáveis e seus métodos. Portanto, o comportamento de um objeto de nível base pode ser facilmente modificado pelo seu meta-objeto.
- *Associação dinâmica*: a associação entre um meta-objeto e um objeto é realizada em tempo de execução. Quando um objeto é instanciado, pode ocorrer também a instanciação de seu meta-objeto. Dependendo do mecanismo de associação, é possível mudar o meta-objeto associado de forma dinâmica. Esta característica permite que um objeto assuma diferentes comportamentos ao longo do processo de execução, dependendo dos meta-objetos associados a ele.
- *Definição circular*: sendo um meta-objeto um objeto, um meta-objeto pode ter associados seus próprios meta-objetos. Assim, é possível construir uma arquitetura de múltiplos níveis reflexivos, denominada, habitualmente, torre de reflexão. Conceitualmente, esta torre pode ser composta de infinitos meta-níveis.

Estas caraterísticas tornam os meta-objetos uma abordagem extremamente adequada para estruturar ferramentas de análise de aplicações, dinamicamente adaptáveis para diferentes funções de análise. Diferentes classes de meta-objetos, fornecendo funcionalidades particulares de análise, podem ser implementadas e estarem disponíveis em uma biblioteca para serem reutilizadas em diferentes ferramentas.

5.1.2.1 Protocolo de meta-objetos

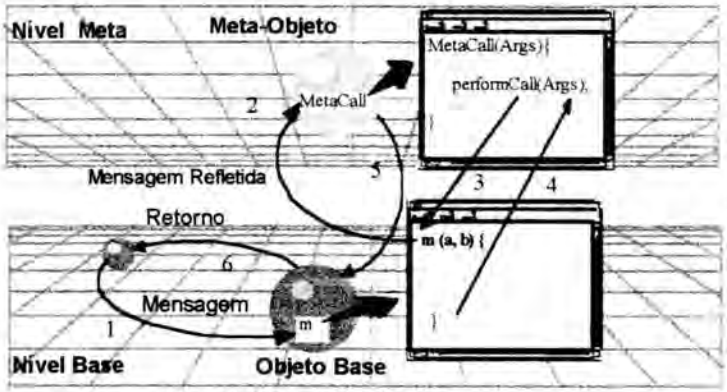
Na provisão de um suporte de meta-objetos surgem, basicamente, três aspectos que devem ser contemplados pela implementação da linguagem:

- transformação da informação do programa necessária para realizar seu tratamento no meta-nível.
- associação entre os objetos do nível base com seus correspondentes meta-objetos.
- a ativação desses meta-objetos quando um objeto do nível base deve realizar alguma operação controlada por um meta-objeto.

A implementação particular destes aspectos fornecido por uma linguagem reflexiva constitui o *protocolo de meta-objetos* (MOP) dessa linguagem.

O primeiro aspecto refere-se aos dados que podem ser manipulados no meta-nível. A atividade computacional do nível base é transformada em dados para o nível superior, determinando quais computações podem ser realizadas. Esta operação de transformação é denominada de reificação [FER 89]. Reificação é a transformação de atributos de um programa em dados disponíveis ao próprio programa. As entidades reificadas constituem a meta-informação sobre a qual realizam-se computações reflexivas. Um protocolo de meta-informações (MIP) tem por objetivo tornar disponíveis ao programador da aplicação, informações a respeito dos componentes do programa que são conhecidas e tratadas normalmente apenas pelo compilador ou interpretador da linguagem de programação. As informações mais comumente disponíveis em um MIP são a classe de um objeto, herança, propriedades dos objetos (estruturas de dados de uma classe e seus métodos) e mensagens entre eles, os quais fornecem a informação relativa à atividade computacional do sistema.

Os dois aspectos restantes referem-se aos mecanismos que definem como a associação entre os dois níveis é implementada e como se inicia o processo de reflexão. Existem, basicamente, duas possibilidades para implementação de computações reflexivas [MAE 88]: a responsabilidade pode ser atribuída ao objeto de nível base, que neste caso contém código mencionando seu meta-objeto, ou pode ser realizada automaticamente pelo sistema de suporte de execução da linguagem. Neste último caso, o



1- A mensagem *m* é enviada a um objeto da aplicação.

2- A mensagem é refletida para o meta-objeto associado com o receptor.

3- O meta-objeto decide exe-cutar o método *m* original, após realizar alguma computação.

4- O método *m* termina sua execução e retorna para o meta-objeto.

5 - O meta-objeto retorna o controle ao nível base, após executar alguma função adicional.

6 - Retorna o controle ao objeto que enviou a mensagem

FIGURA 5.4- Fluxo de controle entre objetos e meta-objetos

meta-objeto é ativado pelo sistema quando ocorre algum evento envolvendo ao seu objeto associado, como uma mudança de estado de variáveis de instância ou o recebimento de uma mensagem dirigida a um método.

O mecanismo mais comum utilizado para ativação de meta-objetos é a interceptação de mensagens entre os objetos do nível base. Na interceptação de mensagens, todas as mensagens enviadas ao objeto, ou mais especificamente, a algum método reflexivo do objeto, são delegadas ao seu meta-objeto, passando este a ser o responsável pelo tratamento da mensagem (FIGURA 5.4).

Uma mensagem pode ser interceptada quando ela é enviada ou quando ela é recebida pelo destinatário. Independente da forma de interceptação, o protocolo de meta-objetos deve, minimamente, fornecer os mecanismos para recuperar a informação relativa ao destinatário da mensagem, o método a ser ativado e os argumentos da mensagem. A reificação destes componentes possibilita que o meta-objeto reenvie a mensagem ao objeto da aplicação, para a execução do método original.

Para realizar a reflexão comportamental de objetos, são necessárias poucas informações sobre a sua estrutura dos objetos: acesso à informações estruturais sobre objetos individuais e reificação de mensagens e métodos na forma de objetos, são suficientes. Seguindo esta linha de reflexão é possível obter distintas granularidades de reflexão comportamental sem interferir no encapsulamento do objeto. A reificação individualizada de uma mensagem que ativa um método e seu conseqüente tratamento pelo meta-objeto, faz com que a reflexão seja restrita ao método chamado, ou seja, torna-se uma reflexão particular de cada método e não de todo o objeto.

CLOS (*Common Lips Object System*) representa o exemplo de MOP mais avançado existente em linguagens atuais. CLOS estende *Common Lisp* para suportar sistemas de objetos baseados em meta-objetos. Através do MOP de CLOS é possível implementar diferentes mecanismos para programação orientada a objetos. A implementação da linguagem representa os principais elementos desta, ou seja, classes, métodos e mensagens, como objetos. O comportamento destes elementos é definido pelo MOP, em termos de meta-objetos que implementam o comportamento de classes, métodos e mensagens.

O funcionamento do sistema de objetos é baseado no conceito de funções genéricas. Uma função genérica é uma função que possui vários corpos diferentes os quais são ativados dependendo do tipo dos argumentos. O MOP especifica um conjunto de funções genéricas definidas por métodos que operam sobre classes. O comportamento implementado nestas funções genéricas determina o comportamento de sistema de objetos.

A programação no meta-nível em CLOS, envolve a definição de novas classes de meta-objetos e a especialização de métodos padrão definidos nas superclasses base do MOP. A meta-classe

de um objeto é a classe de sua classe. A meta-classe determina a representação de instâncias de suas instâncias e os mecanismos de herança, tanto de variáveis como de métodos. O mecanismo de meta-classe é utilizado para prover comportamentos especializados, como por exemplo, tornar persistente o conteúdo de um determinado *slot* definido pela classe. Para implementar este comportamento, uma meta-classe definida pelo usuário deve redefinir o método padrão *slot-value-using-class*, o qual é a função do meta-objeto que determina como o armazenamento de valores em um slot do seu objeto é realizado. Da mesma forma, o MOP pode ser estendido para analisar o código e o comportamento de uma aplicação.

5.2 Protocolos de meta-objetos para linguagens não reflexivas

A capacidade de reflexão é uma propriedade das linguagens que foram projetadas para prover tal habilidade, como por exemplo 3-KRS [MAE 87], ABCL/R [WAT 88], o já descrito CLOS [GAB 91] e, em um grau muito menor, Smalltalk [GOL 83]. No entanto, apesar de uma linguagem não prover capacidades próprias para implementar computações reflexivas, um suporte básico para meta-objetos pode ser agregado através de uma infra-estrutura que implemente os seguintes mecanismos:

- recuperação de meta-informação estática como estrutura interna de instâncias e classes, métodos e herança.
- interceptação das mensagens entre objetos e ativação dos meta-objetos para tratar essas mensagens.
- continuação da execução normal dos métodos refletidos

A complexidade destes mecanismos depende da natureza da linguagem. Smalltalk provê várias facilidades para implementar computações reflexivas [JOH 89], portanto um suporte básico para meta-objetos pode ser implementado mais facilmente.

Em linguagens estáticas e fechadas como C++, este suporte pode ser provido através de um pre-processador, como é o caso de *OpenC++* [CHI 93] ou *Mirror* [ORO 95]. Este pre-processador realiza uma forma de instrumentação de código a qual implementa os mecanismos para associar e transferir o controle aos meta-objetos e gerar a informação estática que pode ser necessária para os meta-objetos. Estes mecanismos podem ser implementados no nível das declarações de classes, não sendo necessário modificar o código dos métodos que implementam a aplicação.

5.2.1 Técnicas de Implementação de MOPs

Um MOP define a associação entre os objetos base e os meta-objetos que sobre eles refletem, e para que estos possam ser ativados, as mensagens dirigidas a aqueles devem ser interceptadas. A seguir discutem-se as principais técnicas para a associação entre objetos e meta-objetos e de interceptação de mensagens.

5.2.1.1 Associação entre objetos e meta-objetos através de herança

Na associação baseada em herança, o objeto herda a habilidade de possuir um meta-objeto associado. A associação pode ser implementada por uma variável de instância que é herdada pelas classes do sistema. Essa técnica encontra restrições em Smalltalk, pois a classe *Object*, da qual os meta-objetos poderiam herdar, é abstrata e não admite definir variáveis de instância.

Uma solução alternativa é implementar as associações através de um dicionário contido numa variável de classe definida na classe *Object*. Esse dicionário poderá conter todas as associações entre objetos e meta-objetos.

Uma outra alternativa seria definir uma classe especial, digamos *ReflectiveObject*, a partir da qual deveriam herdar todas as classes que terão meta-objetos associados. Esta solução elimina o problema descrito acima mas força a todas as classes que terão meta-objetos associados herdarem

dessa classe. Isto poderia ser aceitável se novas classes devem ser desenvolvidas para uma nova aplicação, mas inibe a reflexão de classes existentes que não herdam de *ReflectiveObject*.

5.2.1.2 Associação através de um objeto mediador

Uma alternativa ao uso de herança é transferir a responsabilidade de manter as associações a um outro objeto, chamado objeto *gerenciador*. Um objeto gerente determina como meta-objetos são associados com os objetos do nível base e como esses meta-objetos são ativados, ocultando os mecanismos utilizados para implementar o comportamento reflexivo.

5.2.1.3 Interceptação de mensagens baseada nos objetos

A mais comum técnica de interceptação de mensagens faz uso de *wrappers*. Neste caso, o objeto destinatário original da mensagem é envolvido por outro objeto (o *wrapper*) através de alguma técnica de programação ou compilação. O objeto envolvedor permanece invisível aos emissores da mensagem.

Quando uma mensagem é enviada ao objeto servidor, é recebida na realidade pelo objeto envolvedor, que pode tomar quaisquer medidas que deseje antes de repassar a mensagem ao destinatário original. Quando este último termina sua função, sua resposta pode passar ainda pelo mesmo objeto envolvedor, que também nessa situação, pode efetuar qualquer tratamento especial antes de devolvê-la ao emissor original.

Em Smalltalk, a principal forma de interceptar mensagens é através da redefinição da mensagem *doesNotUnderstand:*(FIGURA 5.5). Normalmente, a mensagem *doesNotUnderstand:* é enviada ao receptor de uma mensagem inválida, ou seja, se o receptor não pode tratar esta mensagem. A mensagem *doesNotUnderstand:* é implementada na classe raiz *Object* do Smalltalk e possui um argumento, *aMessage*. Esta é constituída pelo seletor e os argumentos da mensagem inválida. Essa mesma mensagem pode ser enviada explicitamente através do uso de qualquer variação da mensagem *perform:with:*, a qual é implementada pela classe *Object*.

Este mecanismo tem algumas vantagens e limitações:

- É útil quando um programa é projetado para ter um comportamento reflexivo e o programador pode decidir quando determinado objeto irá refletir em um meta-objeto. Porém não é totalmente adequado quando o comportamento reflexivo deve ser adicionado de forma externa e transparente. Neste caso, as referências ao objeto não são acessíveis, e seria necessário modificar o código da aplicação.
- Não permite interceptar mensagens enviadas a *self* e *super*. Referências ao objeto

FIGURA 5.5- Mecanismo de interceptação baseado em wrappers em Smalltalk

envolvedor são conhecidas apenas externamente ao objeto refletido, que não possui conhecimento sobre a existência do objeto envolvedor. Essa é uma propriedade desejável para ampliar a reutilizabilidade de uma infra-estrutura reflexiva, mas impõe sérias limitações sobre aplicações que necessitam ter conhecimento sobre a troca de mensagens intra-objetos.

- Permite refletir objetos individuais, mas a reflexão de todas as instâncias de uma mesma classe exige a criação de um envolvedor para cada objeto. Os custos de memória poderiam ter um forte impacto negativo sobre a performance geral do sistema.
- Não permite a reflexão seletiva dos métodos. Todas as mensagens enviadas ao objeto refletido seriam desviadas para o meta-objeto quando apenas uns poucos métodos precisariam de reflexão. Isso poderia resultar em custos inaceitáveis à performance do sistema, forçando a adoção de outras soluções.
- Não permite interceptar uniformemente os métodos de classe. As classes em Smalltalk são mantidas em um dicionário global, e são referenciadas pelo nome no código. Alterar esse dicionário produz uma alteração geral no sistema.

5.2.1.4 Interceptação de mensagens baseada nos métodos

Um mecanismo alternativo de interceptação de mensagens é envolver os métodos originais da classe por outros métodos reflexivos. Isto pode ser facilmente obtido em Smalltalk onde a estrutura que define uma classe é acessível como qualquer outro objeto, e os próprios métodos são descritos e tratados como objetos (FIGURA 5.6).

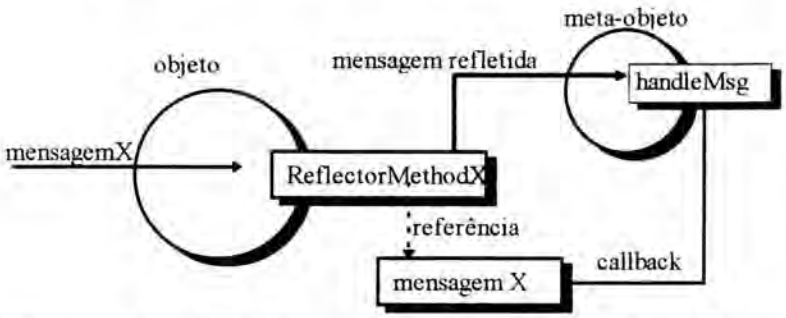


FIGURA 5.6- Esquema da reflexão de mensagens através de métodos refletores em Smalltalk

Em linguagens estáticas como C++ este mecanismo é o implementado, normalmente, pelos pre-processadores. Por exemplo, em Mirror, o pre-processador substitui a classe original por uma outra classe que redefine os métodos a serem refletidos. Estes métodos contêm o código que transfere o controle para os meta-objetos associados com o objeto que recebe a mensagem.

A FIGURA 5.7 mostra um exemplo do processo descrito acima. A definição da classe original é mostrada a esquerda e a classe gerada pelo pre-processador é mostrada a direita. A classe refletida possui os mesmos atributos da classe original, acrescentando alguns outros que implementam o acesso aos atributos originais da classe. Os métodos refletidos são renomeados para conservar a implementação original e novos métodos com o mesmo nome são gerados para realizar a transferência de controle para o meta-nível através da função *MetaCall*. A função *pack* realiza o empacotamento (*marshalling*) dos argumentos.

a)Classe Original

```
class X: public Y {
public:
    int i, j;
    X();

    void f(a,b);


    void g(c);


}
```

b) Classe Refletida

```
    class X: public Y, ReflectiveObject {
public:
        Refl_int i; Refl_int j;
    void __X();
    X();
    void __f(a,b);
    void f(a,b) {reflective ? MetaCall(1,
            pack(a,b)) : __f(a,b); }
    void __g(c);
    void g(c) {reflective ? MetaCall(2,
            pack(c)) : __g(c); }
    }
```

FIGURA 5.7- Substituição de classes em Mirror

Essa implementação tem a desvantagem de ser intrusiva no sentido de que altera a estrutura de uma classe até que os métodos reflexivos sejam removidos. Isso é indesejável quando apenas instâncias específicas deveriam ser refletidas. Também pode ser um problema se a classe é compartilhada entre diversas aplicações, mas pode-se evitar o comportamento reflexivo se o meta-objeto associado simplesmente passar adiante a mensagem.

Ainda assim, oferece diversas vantagens sobre a abordagem baseada nos objetos:

- Intercepta mensagens enviadas a *self* e *super*, as quais são essenciais para analisar a estrutura de especialização de um framework.
- Permite discriminar quais métodos serão refletidos, evitando interceptar todos os métodos de uma classe.
- Permite que diferentes métodos de um mesmo objeto sejam refletidos por diferentes meta-objetos sem a necessidade de programar classes especiais como poderia ser o caso de objetos envolvedores. Também podem ser tratados diferentemente os métodos de instância e de classe.

5.3 Gerenciadores de Meta-Objetos: Adaptabilidade de MOPs

A funcionalidade de um sistema para suporte de reflexão depende em grande medida do tipo da aplicação na qual é requerido. Algumas aplicações podem requerer que cada objeto reflita em seu próprio meta-objeto, outras por sua vez só requerem que alguns métodos sejam refletidos em meta-objetos próprios ou meta-objetos comuns a todas as instâncias de uma classe, por exemplo.

Existem muitas alternativas para dotar a uma linguagem com um suporte de meta-objetos *eficiente* para satisfazer os requisitos de diferentes aplicações que serão estendidas com comportamentos programados no meta-nível, como por exemplo distribuição [JOH 89] [CHI 95]. Entretanto, as implementações propostas na literatura centram-se mais nos aspectos de interceptação de mensagens que nos aspectos da organização e ativação dos meta-objetos. Estes aspectos são muito significativos no caso da utilização meta-objetos para a análise dinâmica de aplicações, pois os meta-objetos não são utilizados para prover nova funcionalidade, mas para prover um mecanismo flexível para implementar diferentes funções de análise do comportamento dessas aplicações. Assim, mecanismos que suportem a implementação flexível da organização e ativação de meta-objetos, apesar de uma perda relativa em termos de eficiência, são necessários. Os *gerenciadores de meta-objetos*, descritos abaixo, são uma alternativa para prover este suporte.

5.3.1 Gerenciadores de Meta-Objetos

Um MOP define um protocolo padrão para a associação de meta-objetos com objetos do nível base e a ativação dos meta-objetos em resposta a eventos produzidos no nível base. Diferentes

linguagens reflexivas provem diferentes MOPs, os quais oferecem mecanismos alternativos para implementar estas funcionalidades, os quais variam no grau de flexibilidade provido. No caso de CLOS, o comportamento operacional do sistema de objetos é implementado através de funções genéricas. Esta abordagem permite a implementação flexível de diversos mecanismos, mas é totalmente dependente da linguagem de suporte (Lisp), e não favorece a composição de comportamento de vários meta-objetos associados com um dado objeto [MUL 95].

Uma abordagem alternativa para resolver estas limitações, mantendo a flexibilidade, são os *gerenciadores de meta-objetos*. Um gerenciador de meta-objetos (doravante MOM do inglês *meta-object manager*) é um objeto que encapsula o comportamento especifico de um dado MOP. Isto é, um MOM define como meta-objetos são *associados* com objetos do nível base e como estes meta-objetos são *ativados*.

Esta abordagem oferece várias vantagens a respeito das ferramentas de análise de programas, a saber:

- ***Políticas de associação***: diferentes MOMs podem encapsular diferentes *políticas de associação* de objetos e meta-objetos. Isto permite separar os aspectos específicos da funcionalidade dos meta-objetos com os aspectos da sua organização em uma meta-arquitetura. Por exemplo, um MOM pode implementar a associação de um único meta-objeto com uma dada classe, de tal modo que esse meta-objeto seja ativado se uma instância não possui seu próprio meta-objeto. Alternativamente, um outro MOM pode permitir associar múltiplos meta-objetos com um dado objeto, ou restringir a associação a um único meta-objeto por objeto.
- ***Estratégias de ativação***: diferentes MOMs podem suportar diferentes *estratégias para a ativação* de meta-objetos. Operacionalmente, um MOM pode agir como um mediador entre os objetos do nível base e os meta-objetos (FIGURA 5.8). As mensagens refletidas desde o nível base são direcionadas para um dado MOM, o qual decide quais meta-objetos, se existe algum, devem ser ativados. Por exemplo, quando vários meta-objetos são associados com um mesmo objeto podem ser utilizadas prioridades de ativação para determinar a ordem na qual esse meta-objetos são ativados. Também é possível implementar mecanismos de combinação como os definidos por CLOS, bem como, mecanismos de composição de meta-objetos como os definidos por Moostrap [MUL 95].

A capacidade para ativar meta-objetos, habilitam um MOM para a suspensão e o reinicio dinâmico da reflexão de mensagens de todos os objetos refletidos, bem como de objetos individuais. Esta capacidade permite, por exemplo, que o usuário determine interativamente quando uma ferramenta deve ou não ativar meta-objetos. Também diferentes funções do meta-nivel podem ser ativadas ou desativadas sem necessidade de acessar meta-objetos específicos como seria o caso de implementações baseadas só no conceito de meta-objeto. Em mecanismos baseados em herança, os meta-objetos não são diretamente acessíveis, portanto, esta funcionalidade não pode ser facilmente alcançada. Neste caso seria necessário manter as referencias a todos os meta-objetos da aplicação, e esta solução é semelhante a ter um objeto gerenciador.

Utilizando MOMs o processo de ativação de meta-objetos é dividido em duas fases:

Uma mensagem é refletida por um objeto do nível base para um dado MOM.
Fase 1: *Procura de meta-objetos*.
O MOM procura meta-objetos associados com o objeto do nível base.
Fase 2: *Ativação de meta-objetos*.
Se algum meta-objeto foi encontrado e a reflexão de mensagens está ativa
então transfere o controle para esses meta-objetos de acordo
com a política de ativação implementada pelo MOM.
caso contrário continua com a execução normal do método refletido.

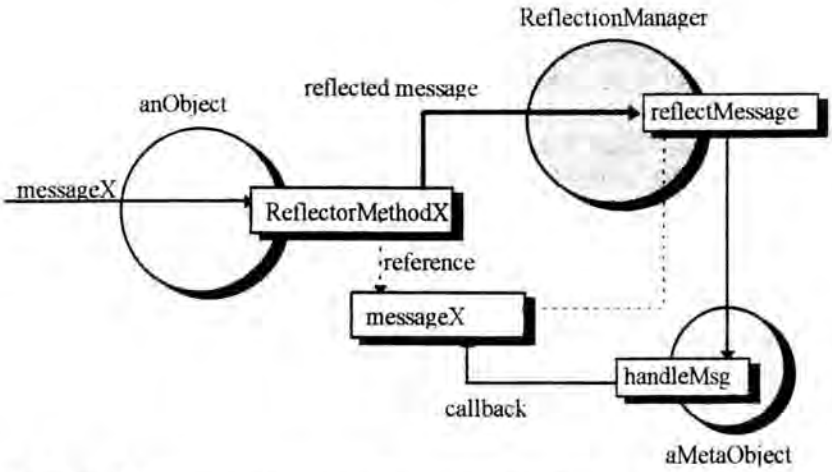


FIGURA 5.8- Relacionamento entre objetos e meta-objetos através do gerenciador

Do ponto de vista da construção de ferramentas os MOMs oferecem a vantagem de prover um interface de alto nivel para organizar os meta-objetos de forma independente da funcionalidade implementada por eles. A separação do mecanismo de associação permite especializá-lo para implementar serviços de gerenciamento específicos para as necessidades da ferramenta, como por exemplo os mecanismos de substituição de meta-objetos necessários para a abordagem orientada pela arquitetura, descrita no capítulo anterior.

O conceito de MOM é o fundamento através do qual um framework para suporte de meta-objetos pode ser implementado, tal com é descrito no capitulo seguinte.

6 O Framework Luthier

No capitulo 4, analisaram-se as diferentes atividades que caraterizam as ferramentas de apoio à compreensão de software: captura, representação, análise de abstrações e apresentação, e exploração, da informação. Dentro de cada uma destas atividades, identificaram-se os requisitos essenciais das técnicas de suporte envolvidas na construção de ferramentas de apoio na compreensão de software orientado a objetos, descrevendo, sinteticamente, as abordagens adotadas neste trabalho.

Como conclusão principal da análise efetuada, surgiu que a provisão de mecanismos que permitam adaptar as ferramentas para as caraterísticas dinâmicas proprias dos programas orientados a objetos, bem como para a produção de diferentes visualizações com capacidades de navegação e manipulação direta, são requisitos essenciais que devem ser contemplados. Estes requisitos de adaptabilidade conduzem ao projeto de um framework que forneça os mecanismos genéricos básicos para as funcionalidades seguintes:

- Exploração iterativa do framework através de múltiplas visões, tanto no nível da organização de classes quanto no nível de comportamento das instâncias.
- Utilização de técnicas de reflexão computacional baseadas em meta-objetos, adaptadas para linguagens não reflexivas, para a adaptabilidade dinâmica dos mecanismos de captura de informação do programa analisado.
- Organização da informação coletada em termos de um modelo de hipertexto avançado.
- Suporte de livros de projeto.
- Interfaces de manipulação direta que habilitem a seleção e o *zooming* das apresentações visuais, fornecendo mecanismos alternativos de exploração.
- Mecanismos de geração de abstrações, filtragem e seleção de informação independentes das visualizações.

O framework Luthier foi projetado, e desenvolvido em Smalltalk-80, para prover um suporte extensivel, e dinamicamente adaptável por composição de objetos, para satisfazer estes requisitos. Luthier compõe-se de quatro sub-frameworks, que abrangem a funcionalidade necessária para as quatro atividades citadas:

- ***LuthierMOPs***: Suporte de meta-objetos baseado no conceito de *gerenciadores de meta-objetos*, para captura de informação.
- ***LuthierBooks***: Extensão do framework de suporte de gerenciamento de hiperdocumentos, PROMETO [ORT 95], para a representação da informação capturada e gerenciamento de livros persistentes de projeto.
- ***LuthierAbstractors***: Suporte genérico para a construção de abstrações e escalas de abstração.
- ***LuthierViews***: Extensão do framework MVC para a construção de interfaces com o usuário de manipulação direta e visualizações da informação coletada.

Este capitulo tem por objetivo descrever as carateristicas relevantes de Luthier, tanto de uma perspectiva informal de sua estrutura essencial e os mecanismos de instanciação de funcionalidade, quanto da especificação formal do projeto, em termos de contratos [HEL 90].

A descrição organiza-se da seguinte maneira: em primeiro lugar descreve-se a estrutura global do framework e sua utilização para construção de ferramentas, a seguir, cada sub-framework componente é descrito de acordo com a estrutura:

1. Descrição informal da funcionalidade do sub-framework através de diagramas de estrutura de objetos, grafos abstratos de interação e padrões de instanciação de funções relevantes, com exemplos de código que ilustram a formas comuns de instanciação.
2. Especificação formal das colaborações baseada na notação de contratos [HEL 90], extendida conforme descrito no Anexo 1.
3. Definição das interfaces das classes principais e sua participação nos contratos especificados na seção previa.

6.1 Luthier: Caraterísticas Gerais

Luthier tem por objetivo fornecer um suporte adaptável, principalmente através de composição de objetos, para implementar mecanismos especializados para a construção de ferramentas de análise e visualização da estrutura estática e dinâmica de programas orientados a objetos.

Uma ferramenta típica construída com Luthier estará composta por um conjunto de meta-objetos que monitoram a execução de uma aplicação, gerando uma representação da informação requerida utilizando um gerenciador de hiperdocumentos. O sistema de visualização requisitará, ao nível inferior, a informação a ser visualizada, a qual será provida por abstratores encarregados de selecionar ou construir abstrações da informação contida na representação de hiperdocumento.

Através do sub-framework de suporte de meta-objetos é possível desenvolver diferentes funcionalidades de análise e monitoração de um programa, através da implementação de meta-objetos específicos. Dependendo da ferramenta, um gerenciador de meta-objetos especializado pode ser desenvolvido para coordenar a atividade dos diferentes meta-objetos.

LuthierBooks provê o suporte para definir representações especificas da informação coletada em termos de classes de nodos, simples e compostos, e classes de elos. Esta representação pode ser gerada pelos meta-objetos que capturam a informação do programa e ser armazenada de forma persistente, para formar parte de livros de documentação. Os livros de documentação, são implementados em termos do mesmo suporte de hipertexto, habilitando a construção de bibliotecas de projeto organizadas como um hiperdocumento.

LuthierViews define uma infra-estrutura para implementar visualizações com interface de manipulação direta, dinamicamente reconfiguráveis. Este sub-framework é uma extensão do framework MVC provido pelo ambiente de implementação VisualWorks, cujas principais caraterísticas foram descritas nos capítulos 2 e 3.

A construção de visualizações implica, habitualmente, no desenvolvimento de subclasses que devem fornecer a implementação de poucos métodos encarregados de produzir a representação gráfica dos objetos a serem visualizados, e sua composição, no caso de objetos visuais compostos. A construção da interface externa de uma ferramenta é realizada utilizando o editor de interfaces provido pelo MVC. Isto simplifica em muito a definição da estrutura global da interface, embora limite as possibilidades de generalização de comportamento pelas caraterísticas do seu projeto.

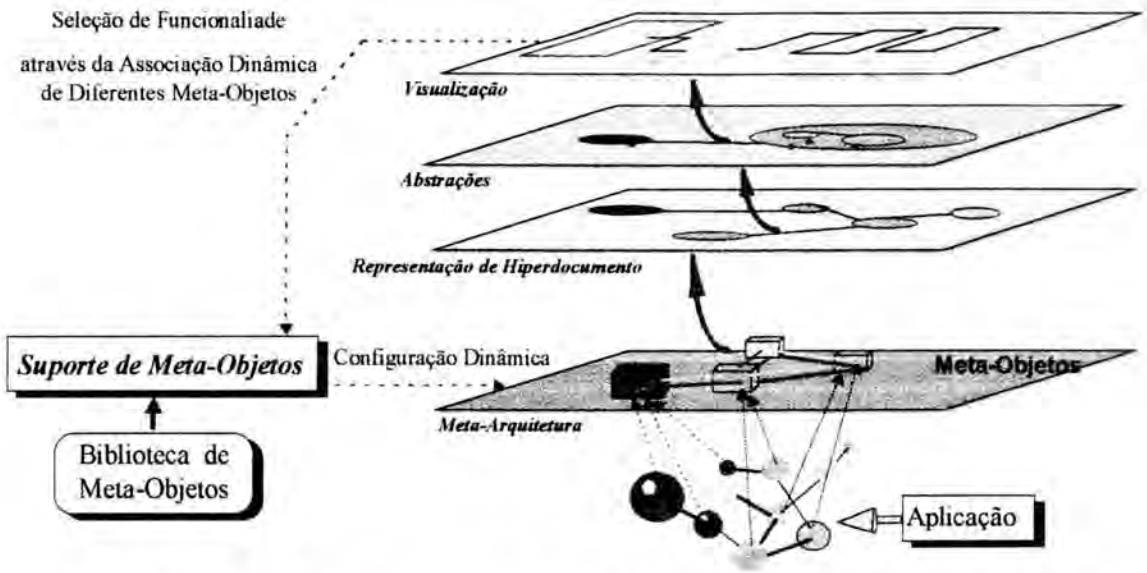


FIGURA 6.1- Estrutura genérica de uma ferramenta construida com Luthier

LuthierViews fornece, também, um conjunto de componentes para a criação de livros com texto formatado, em base a estilos definíveis pelo usuário, capacidade de edição de texto e inserção de componentes visuais gerados pelas visualizações e, até mesmo, ferramentas de visualização, como parte integrante de uma pagina de um livro normal. Estas funcionalidades permitem aumentar as possibilidades de organização e apresentação visual da informação de um programa analisado.

LuthierAbstractors provê, essencialmente, a funcionalidade genérica que habilita o gerenciamento automático de mecanismos de *zoom semântico* das visões, guiado por escalas de abstração definiveis externamente. Os abstratores representam mais um conceito abstrato de projeto que uma estrutura generalizável, razão pela qual, é necessário programar subclasses específicas para cada tipo de funcionalidade desejada. Entretanto, os abstratores são o conceito chave que habilita uma fatoração de funcionalidade que torna as visões e a representação mais simples e reutilizáveis. Também, eles representam um ponto de extensão de funcionalidade para implementação de diferentes algoritmos de análise e reconhecimento de abstrações, filtragem e seleção de informação, que pode ser integrado transparentemente com qualquer visualização desenvolvida. Por exemplo, algoritmos de reconhecimento de padrões ou subsistemas podem ser implementados como métodos específicos de um abstrator.

Um aspecto importante que deve ser destacado é que, a nivel abstrato, a interdependência entre os quatro sub-frameworks é mínima, pudendo, então, serem utilizados em forma isolada para diferentes aplicações. Deste modo, a estrutura fornecida por Luthier pode ser considerada como um *toolkit* extensivel de funcionalidades básicas. É possível, por exemplo, construir ferramentas de animação de algoritmos que utilizem meta-objetos que monitoram e informam ao sistema de visualização dos eventos ocorridos no programa, bem como, implementar visualizações independentes da informação contida na representação, como é o caso de livros de documentação.

A FIGURA 6.2 apresenta a estrutura global de Luthier, utilizando a notação de grafo de colaborações [WIR 90], na qual as colaborações entre os subsistemas, representam relacionamentos potenciais que aparecem freqüentemente entre os componentes das ferramentas desenvolvidas.

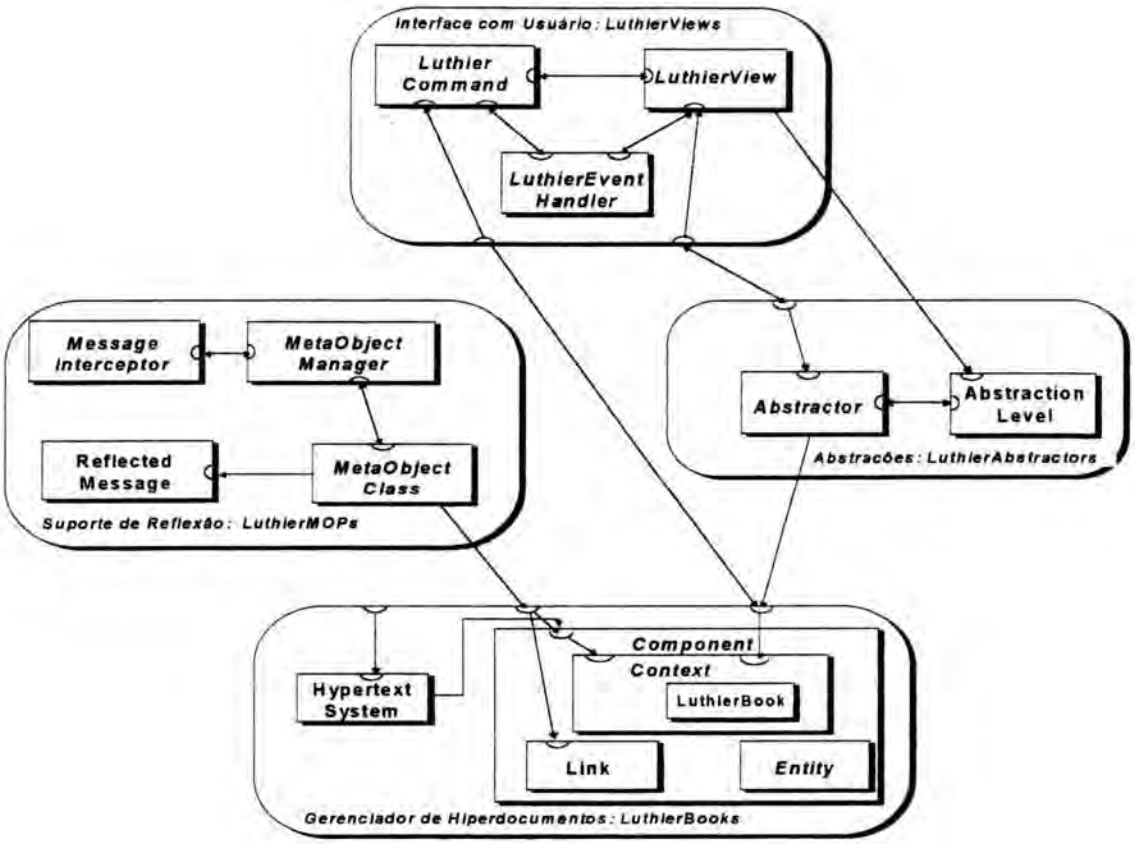


FIGURA 6.2- Diagrama de colaborações de Luthier

6.2 *LuthierMOPs*: O sub-framework de meta-objetos

O sub-framework para suporte de meta-objetos, LuthierMOPs [CAM 96], provê uma infra-estrutura flexível para a associação de meta-objetos com classes, instâncias ou métodos específicos. Um meta-objeto pode ser associado a um método, a um conjunto de métodos (ainda de classes diferentes), a uma instância ou um conjunto de instâncias de diferentes classes; a uma classe ou um conjunto de classes. Um método, instância ou classe pode ter vários meta-objetos associados.

6.2.1 Estrutura geral

LuthierMOPs se baseia no conceito de gerenciadores de meta-objetos (descrito no capítulo anterior), para fornecer um mecanismo extensível e adaptável para a implementação de protocolos de meta-objetos. Um gerenciador de meta-objetos contém a informação necessária para determinar qual meta-objeto deve ser ativado quando uma mensagem é refletida. A FIGURA 6.3 apresenta o diagrama de objetos do sub-framework. Este é composto por quatro classes principais: *MetaObjectManager*, *MetaObjectClass*, *ReflectedMessage* e *MessageInterceptor*.

A interação abstrata entre estas classes é apresentada na FIGURA 6.4. Quando um objeto refletido recebe uma mensagem, esta mensagem é desviada pelo mecanismo de interceptação para o gerenciador associado. O gerenciador procura meta-objetos associados com o objeto que refletiu a mensagem (através da mensagem *findMetaObjectsFor:*) e decide quando transferir o controle para esses

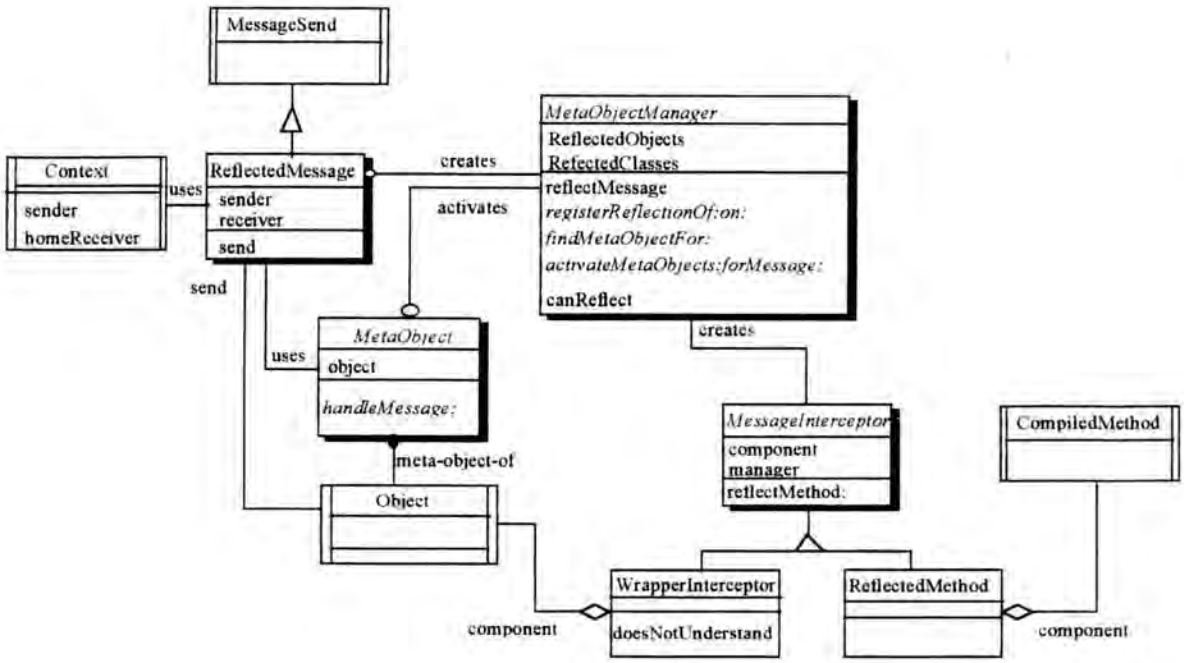


FIGURA 6.3- Modelo de objetos de LuthierMOPs

meta-objetos (através da mensagem *activateMetaObejcts:*) enviando-lhes a mensagem *handleMessage: manager:*. Quando ativado, um meta-objeto recebe uma instância da classe *ReflectedMessage*, a qual contém a informação relativa a mensagem refletida. O meta-objeto pode executar o método original enviando a essa instância a mensagem *send*.

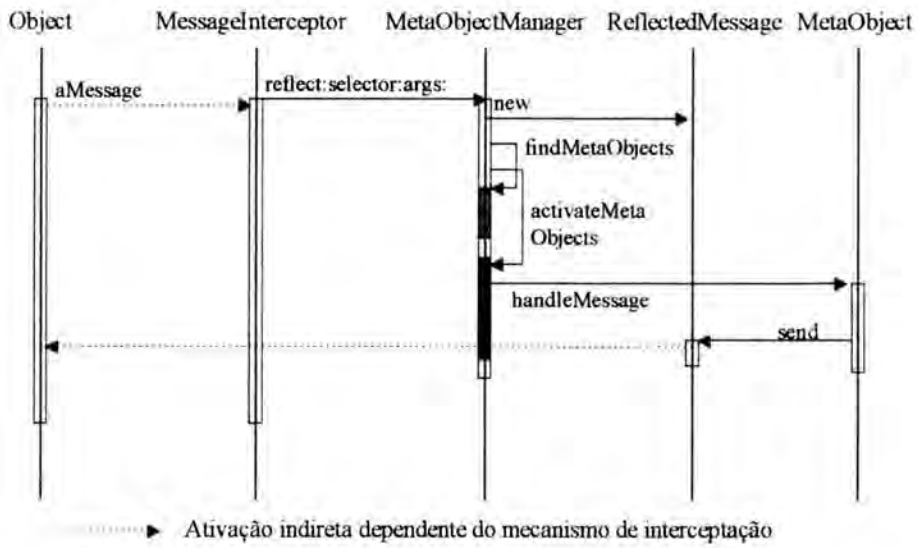


FIGURA 6.4- Fluxo abstrato de controle de LuthierMOPs

6.2.2 Definição de meta-objetos

O comportamento genérico dos meta-objetos é definido pela classe abstrata *MetaObjectClass*. A funcionalidade básica de um meta-objeto é tratar as mensagens recebidas pelo(s) objeto(s) que refletem seu comportamento nele.

O protocolo para o tratamento de mensagens definido por *MetaObjectClass* consiste do método abstrato *handleMessage:manager:*, o qual é chamado por um gerenciador quando uma mensagem foi refletida. Subclasses especificas de *MetaObjectClass* devem prover a implementação deste método para tratar as mensagens refletidas por seu(s) objeto(s) associado(s).

Os argumentos do método *handleMessage:manager:* são uma instância da classe *ReflectedMessage*, o qual contém a informação acerca da mensagem refletida, e o gerenciador que ativou o meta-objeto. Um meta-objeto é livre de enviar a mensagem ao objeto receptor original ou a qualquer outro objeto que deva ser ativado.

Exemplos de Instanciação:

- Uma classe *TracerObjectClass* é projetada para mostrar na tela as mensagens recebidas por um objeto, segundo o formato seguinte:

  classe origem --> seletor classe receptora (classe implementadora)

O método *handleMessage: manager:* recebe a instância de *ReflectedMessage* e interage com ela para obter a informação acerca da classe do originador da mensagem, a classe do receptor e a classe na qual a implementação da mensagem foi encontrada (mensagens *senderClass*, *receiverClass* e *homeReceiver Class*).

```
MetaObjectClass subclass: #TracerObjectClass
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Reflection Library'!

handleMessage: msg manager: aManager
    Transcript show: msg senderClass name, '-->',
                     msg selector ' - ',
                     msg receiverClass name asString '( ',
                     msg homeReceiverClass name,')'.
    ^msg send.
```

- Um outro exemplo de especialização do método *handleMessage:manager:* é um meta-objeto projetado para coletar as instâncias que recebem e enviam uma dada mensagem, as quais são registradas pelo gerenciador associado ao meta-objeto através do método *registerSender*

```
MetaObject subclass: #InstancesCollectorMetaObject

.....

handleMessage: aReflectedMessage manager: aManager
        "Checks if the message is a request to open the InstancesBrowser"
        aReflectedMessage selector == #inspectInstances
         ifTrue:[ ^aManager inspectInstances].
        ifFalse:[    "Register instance information and execute the reflected method"
                aManager registerSender: aReflectedMessage.
                ^ aReflectedMessage send.]
```

Este exemplo mostra como um meta-objeto pode estender o comportamento de seu objeto associado. O método *inspectInstances* implementado neste exemplo pelo gerenciador, abre um inspetor sobre a lista de instâncias coletada. Se o seletor da mensagem refletida é diferente de *inspectInstances*, o meta-objeto registra a mensagem no gerenciador e executa o método original.

Funcionalidades Adicionais

- Inicialização parametrizada

Quando um meta-objeto é programado para implementar alguma funcionalidade que depende de parâmetros específicos, o método *initializeWith:* é provido para especificar um bloco de código que implementa a inicialização dos parâmetros de cada meta-objeto. O bloco recebe um parâmetro que é o meta-objeto a ser inicializado. Por exemplo:

```
anyManagerClass reflectEachInstanceOf: aClass
                on:MultipleInheritanceMeta
                initalizingBlock:[:mo| mo superclasses: #( Class1 Class2 Class3)].
```

produz que cada instância criada de um meta-objeto que implementa herança múltipla receba a lista de superclasses da classe associada. Isto evita a programação de mecanismos especiais de inicialização, tanto no meta-objeto quanto nos gerenciadores.

- Controle do Processo de Reflexão

Alguns meta-objetos são projetados para acrescentar funcionalidades especiais a um conjunto de classes, ainda mais, tratando a reflexão só métodos específicos dessas classes. Por esta razão *MetaObjectClass* provê mecanismos para specificar quais classes e métodos cada classe de meta-objeto pode tratar. O método hook *reflectedClassesAllowed*, pode retornar um dicionario com as classes e métodos dessas classes que podem ser tratadas por cada classe específica de meta-objeto. Por exemplo, a classe *InteractiveReflectionActivator* é definida para interceptar só o método *setStateFromVector:* da classe *InputState*, portanto redefine define o método:

```
reflectedClassesAllowed
    ^Dictionary withKeysAndValues: (Array with: InputState
                                    with: (OrderedCollection with: #setStateFromVector:))
```

O comportamento por falta deste método é retornar *nil*, o qual não estabelece nenhuma restrição.

6.2.3 Implementação de MOPs

O mecanismo genérico dos gerenciadores de meta-objetos é implementado pela classe abstrata *MetaObjectManager*. Esta classe deve ser especializada para implementar mecanismos alternativos de associação e ativação de meta-objetos. Subclasses de *MetaObjectManager* implementam diferentes MOPs fornecendo a implementação dos métodos abstratos definidos para:

- associação de objetos com meta-objetos
- seleção de meta-objetos associados com um objeto que refletiu uma mensagem
- transferir o controle para esses meta-objetos.

Os detalhes deste mecanismos são descritos a seguir.

6.2.3.1 Associação de meta-objetos com objetos da aplicação

MetaObjectManager define a interface para converter métodos e objetos em reflexivos e a associação de meta-objetos com esses objetos.

O método abstrato ***registerReflectionOf:on:*** deve ser implementado por subclasses para criar a associação entre objetos do nível base e os meta-objetos. Este método é chamado pelos métodos fornecidos que implementam as diferentes alternativas de reflexão de métodos de instância e de classe

Exemplos de Instanciação

- A ferramenta para apoio à compreensão de frameworks descrita no próximo capítulo [CAM 95][CAM 96] utiliza uma subclasse de *MetaObjectManager*, *LayeredManager*, que associa meta-objetos por falta com cada classe que será monitorada por meta-objetos. Todas as instâncias de uma classe monitorada refletem sobre um de esses meta-objetos. Para implementar esta associação *LayeredManager* utiliza um dicionário que contem, para cada classe refletida, uma coleção de meta-objetos. Esta coleção é ordenada pela ordem decrescente de um número de prioridade de cada meta-objeto. Estas prioridades são utilizadas para especificar a ativação dos meta-objetos (capacidade semelhante às regras de combinação de métodos de CLOS). LayeredManager redefine o método *registerReflectionOf:on:* para agregar meta-objetos a seu dicionário conforme mostrado abaixo:

```
MetaObjectManager subclass: #LayeredManager
.....
registerReflectionOf: aClass on:aMetaObject
        "O dicionario ReflectedClasses contém uma entrada para cada classe refletida com uma
        coleção dos meta-objetos associados com essa classe"
        |metaObjects|
        metaObjects := ReflectedClasses at: aClass
                ifAbsent: [ReflectedClasses at: aClass
                 put: (SortedCollection sortBlock:[ :x :y | x priority > y priority])]
        metaObjects add: aMetaObject
```

A instância de *SortedCollection* recebe um bloco especificando um predicado de ordenamento que será utilizado para ordenar os elementos na coleção. A mensagem *priority* deve ser redefinida por cada classe específica de meta-objeto para retornar um número de prioridade associado com a classe.

- Um outro exemplo pode ser uma subclasse de gerenciador, *SingleMetaObjectManager*, que associe a referencia ao meta-objeto com o método que reflite nesse meta-objeto, permitindo encontrar o meta-objeto associado de forma mais eficiente. A associação entre meta-objeto e o método pode ser implementada da seguinte maneira:

```
MetaObjectManager subclass: #SingleMetaObjectManager
.....
registerReflectionOf: aReflectedMethod  on:aMetaObject
        " ReflectedMethod will contain the reference to the meta-object"
        aReflectedMethod metaObject: aMetaObject.
```

6.2.3.2 Ativação de meta-objetos

A classe *MetaObjectManager* implementa o mecanismo genérico para ativação de meta-objetos, quando uma mensagem é refletida. Subclasses de *MetaObjectManager* devem prover a implementação dos métodos seguintes:

- *findMetaObjectsFor:* para encontrar o(s) meta-objeto(s) associados com o receptor da mensagem.
- *activateMetaObject:withMessage:* para transferir o controle para esse(s) meta-objeto(s).

O método template *reflectMessage:args:forObject:* implementa o algoritmo genérico que maneja o processo de encontrar, e ativar, aqueles meta-objetos associados com o objeto que refletiu uma mensagem. Este método é chamado pelo mecanismo de interceptação de mensagens.

```
reflectMessage: selector args: arguments forObject: anObject
" Implements the generic algorithm for the activation of those meta-objects associated to the
reflector object. "
    |mo |
        "Tests manager's reflection state"
    self canReflect
        ifFalse:[ ^ self callBackMethod: selector args: arguments forObject: anObject ].

    aReflectedMessage:= self defaultReflectedMessage selector:selector arguments:
                                                    arguments receiver: anObject
                                                    sender: thisContext sender receiver.
        "Search the metaobjects associated with the reflector object"
    mo:= self findMetaObjectsFor: aReflectedMessage.
    mo isNil
        ifTrue:[  " There are no metaobjects. Evaluate the original method"
                ^self callBackMethod: selector args: arguments forObject: anObject]
        "Activate the meta-objects"
    ^self activateMetaObject: mo withMessage: aReflectedMessage
```

Exemplos de Instanciação:

Para implementar a ativação de meta-objetos, cada tipo de gerenciador deve implementar os métodos abstratos *findMetaObjectsFor:* e *activateMetaObject:withMessage:* de acordo com sua política. Por exemplo, quando uma mensagem é refletida, *LayeredManager* ativa o meta-objeto da classe na hierarquia na qual o método foi encontrado. Desta forma, cada meta-objeto trata só com as mensagens implementadas por sua classe associada. A implementação destes métodos é a seguinte:

```
findMetaObjectsFor: aReflectedMessage
    |mo|
    mo:= ReflectedClasses at: aReflectedMessage homeReceiverClass ifAbsent:[]]
    ^mo
```

```
activateMetaObject: aCollection withMessage: aReflectedMessage
        "Activates all the associated meta-objects "
    aCollection do:[ :mo| mo handleMessage: aReflectedMessage].
        "Returns the value stored by the message"
    ^aReflectedMessage value.
```

No caso do gerenciador *SingleMetaObjectManager*, que associa a referencia ao meta-objeto com o método que reflite nesse meta-objeto, a implementação de ambos métodos é a seguinte:

```
findMetaObjectsFor: aReflectedMessage
    |mo|
    ^aReflectedMessage reflectorMethod metaObject.

activateMetaObject: aMetaObject withMessage: aReflectedMessage
        "Activates the associated meta-object "
    aMetaObejct handleMessage: aReflectedMessage.
        "Returns the value stored by the message"
    ^aReflectedMessage value.
```

Através destes simples mecanismos é possível implementar diferentes estratégias de seleção e ativação, de acordo com as necessidades de cada ferramenta específica, sem necessidade de modificar o comportamento dos meta-objetos existentes. É necessário aclarar aqui que estas estratégias de implementação são possíveis devido á natureza polimórfica pura de Smalltalk, implementações em linguagens tipadas podem requerer refinar o protocolo com sobrecarga de operadores, como por exemplo, C++.

6.2.4 Adaptação da reflexão e reificação de mensagens

O mecanismo genérico de ativação de meta-objetos provido por *MetaObjectManager*, descrito acima, fornece quatro pontos de adaptação de funcionalidade:

- *canReflect* : informa se o gerenciador deve ativar ou não meta-objetos
- *startReflect*: habilita a ativação de meta-objetos
- *stopReflect*: inibe a ativação de meta-objetos
- *defaultReflectedMessage*: retorna a classe que reifica a mensagem refletida

A implementação por falta do método *canReflect*, retorna o valor da variável *Reflecting*, definida por *MetaObjectClass*. Quando o valor desta variável é falso, a mensagem interceptada é enviada diretamente ao objeto receptor original, como se na realidade não existissem meta-objetos. Um cliente pode alterar o estado de reflexão do gerenciador através das mensagens *startReflect* e *stopReflect*.

A implementação por falta de *defaultReflectedMessage* retorna a classe *ReflectedMessage*. Gerenciadores específicos podem redefinir este método para retornar uma subclasse que defina um comportamento especializado das mensagens refletidas.

Exemplo de Instanciação:

O meta-objeto *InteractiveReflectionActivator* é projetado para habilitar o inibir a reflexão de mensagens quando uma combinação de teclas é pressionada. Este meta-objeto é associado com a instância da classe *InputState*, a qual processa os eventos de entrada.

```
MetaObjectClass subclass: #InteractiveReflectionActivator
.....
handleMessage: msg
        |anState|
        msg send.
        anState := msg receiver.
        anState ctrlDown & anState shiftDown
                ifTrue:[MetaObjectManager stopReflect].
        anState ctrlDown & anState shiftDown
                ifTrue:[MetaObjectManager startReflect]
```

6.2.5 Reificação de mensagens

A reificação das mensagens trocadas entre objetos é um requisito fundamental de um suporte para implementação de meta-objetos. Por reificação de uma mensagem entende-se a transformação da informação relativa a uma mensagem, como seletor, argumentos, receptor, etc., em um objeto que pode ser tratado como um objeto normal no meta-nível. A classe *ReflectedMessage* implementa esta funcionalidade.

Instâncias desta classe são criadas quando uma mensagem é refletida. Uma instância de *ReflectedMessage* encapsula toda a informação relativa à mensagem refletida como seletor, argumentos, receptor e originador. Um meta-objeto pode executar o método original enviando a mensagem *send* para a instância de *ReflectedMessage* que recebe para ser tratada.

Exemplos de Instanciação:

No caso geral, a classe *ReflectedMessage* não precisa ser redefinida. A implementação do método *send*, fornecida por falta, retém o valor da primeira execução e retorna este valor quando vários meta-objetos tentam executar o mesmo método refletido:

```
send
        ^value isNil ifTrue:[ value:= self execute] ifFalse:[value].
```

Caso este comportamento não seja desejado pode-se criar uma subclasse que redefina este método.

6.2.6 Interceptação de mensagens

A interceptação de mensagens é o aspecto central da implementação do suporte de meta-objetos. A classe abstrata *MessageInterceptor* define a interface geral para interceptar mensagens, de acordo com os dois métodos discutidos no capítulo anterior: baseada nos métodos e baseada nos objetos. O primeiro caso é implementado pela subclasse *ReflectedMethod* e o segundo pela subclasse *WrapperInterceptor*.

A interface para a criação de interceptores é:

```
onSelectors:({all | Lista de Seletores} ) in: ({Object|Classe}) reflectingOn: (aMetaObjectManager)
```

Entretanto, esta interface não é, no caso geral, visível para o usuário do framework pois é utilizada pela interface de reflexão provida pela classe *MetaObjectManager*.

6.2.6.1 Interceptação baseada nos métodos

A classe *ReflectedMethod* implementa o comportamento para encapsular métodos e transferir o controle para um dado gerenciador. A reflexão de um método é implementada (em Smalltalk) substituindo, no dicionário de métodos da classe, a instância de *CompiledMethod* que contém o código do método, por uma instância de *ReflectedMethod*. A instância de *ReflectedMethod* mantém a referência ao método original, a qual será usada, posteriormente, quando um meta-objeto decide executar a implementação original do método refletido.

ReflectedMethod fornece a implementação do método de classe encarregado de substituir um método normal por um método que transfere o controle para um gerenciador dado:

```
reflect: aCollection of: aClass manager: aManager
    | m reflectedMethod |
        "Recovers the method from the class dictionary"
    aCollection do:[ :aSelector|
        m := aClass compiledMethodAt: aSelector ifAbsent: [^nil].
        "Tests if it was already reflected"
        m isReflector ifTrue: [self unReflectedMethod:m].
        "Creates the reflected method and changes the method dictionary"
        aClass addSelector: aSelector
            withMethod: (reflectedMethod := self selector: aSelector
                                                 class: aClass
                                                 reflectingOn: aManager).
        reflectedMethod component: m].
```

A instância de *ReflectedMethod* contém o código compilado que implementa a transferência de controle desde um receptor da mensagem para o gerenciador dado. Por exemplo, a reflexão de um método chamado de *setStateFromVector:*, resultará em um método com a mesma assinatura, e um corpo que envia a mensagem *reflectMessage:args:forObject:*. para o gerenciador associado:

```
setStateFromVector: arg
    ^manager reflectMessage: #setStateFromVector: args:(Array with:arg) forObject:self).
```

Exemplo de Instanciação:

A classe *MetaObjectManager* fornece a interface para produzir a interceptação de todos os métodos de classe ou instância, bem como métodos específicos, utilizando métodos refletores. Por exemplo:

- Reflexão dos métodos de instância da classe *InputSate* sobre uma instância de *TracerObjectClass*.

```
MetaObjectManager reflectInstancesOf: InputState
                  on: TracerObjectClass.
```

- Reflexão dos métodos de classe da classe *InputSate* sobre uma instância de *TracerObjectClass*.

```
MetaObjectManager reflectClass: InputState
                  on: TracerObjectClass.
```

- Nos dois últimos casos os meta-objetos são criados pelo gerenciador, portanto, ambos tipos de métodos reflitem sobre diferentes meta-objetos. Para refletir ambos tipos de métodos sobre o mesmo meta-objeto deve ser utilziada a mensagem *reflect...upon:*. No exemplo abaixo os métodos de instância e de classe de *InputState* são refletidos sobre um meta-objeto da classe *TracerObjectClass*.

```
metaObject := TracerObjectClass new.
MetaObjectManager reflectInstancesOf: InputState
                                upon: metaObject.
MetaObjectManager reflectClass: InputState
                          upon: metaObject.
```

- Reflexão de cada instância da classe *InputSate* sobre uma instância de *TracerObjectClass*. Isto produzirá que um novo meta-objeto seja associado com cada nova instância que seja criada.

```
MetaObjectManager reflectEachInstanceOf: InputState
                              on: TracerObjectClass.
```

- Reflexão do método *setStateFromVector* da classe InputState sobre uma instância da classe de meta-objeto *InteractiveReflectionActivator*

```
MetaObjectManager reflectMethod: #setStateFromVector:
                                of: InputState
                                on:InteractiveReflectionActivator new.
```

6.2.6.2 Interceptação baseada nos objetos

A classe *WrapperInterceptor* fornece o suporte para interceptar mensagens recebidas por instâncias específicas de classes que não podem, ou não precisam, ser refletidas utilizando métodos refletores. Essencialmente, uma instância de *WrapperInterceptor* implementa o mesmo comportamento dos métodos refletores, mas a reflexão de mensagens é implementada através da redefinição do método *doesNotUnderstand* (esta mensagem é enviada pelo suporte de execução de Smalltalk quando um objeto não implementa uma dada mensagem).

A especialização do método *reflect: aCollection of: anObject manager: aManager* provida por *WrapperInterceptor* faz uso do mecanismo de troca de referências de objetos, ***become:***, provido por Smalltalk. Através deste mecanismo, as referencias do interceptador e do objeto interceptado são trocadas, de modo tal que todo objeto que possua uma referência ao objeto interceptado, referenciará ao interceptador. Deste modo, consegue-se uma reflexão transparente para os clientes do objeto interceptado.

```
reflect: aCollection of: anObject manager: aManager
    " Creates a wrapper interceptor for a metaobject for anObject"
    methods := aCollection.
    self manager: aManager.
    self component: anObject.
    self become: anObject.
```

A reflexão de mensagens é realizada através da redefinição do método *doesNotUnderstand: aMessage*, a qual verifica se a mensagem não compreendida deve ser refletida para transferir o controle ao manager.

```
doesNotUnderstand: aMessage
    "Tests if the wrapper must intercept the message"
    (methods includes: aMessage selector )
            ifTrue: [^manager reflectMessage: (WrappedMessage
                                message: aMessage
                                forObject: object].
```

```
"Continue the normal execution of the method"
^component perform: aMessage selector withArguments: aMessage arguments
```

Exemplo de Instanciação:

A classe *MetaObjectManager* fornece a interface para produzir a interceptação de métodos de instância ou métodos específicos, utilizando interceptação com *wrappers*. Por exemplo:

- Reflexão utilizando interceptação baseada nos objetos do método *setStateFromVector* de uma instância da classe InputState sobre uma instância da classe de meta-objeto *InteractiveReflection Activator*

```
MetaObjectManager wrapperReflect:#setStateFromVector
                        object: anInputState
                        metaObject: InteractiveReflectionActivator new
```

- Reflexão utilizando interceptação baseada nos objetos de cada instância da classe *InputSate* sobre uma instância de *TracerObjectClass*. Isto produzirá que um novo meta-objeto seja associado com cada nova instância que seja criada.

```
MetaObjectManager wrapperReflectEachInstanceOf: InputState
                        on: TracerObjectClass
```

6.2.7 Especificação de contratos

6.2.7.1 Mecanismo de reflexão de métodos e associação de meta-objetos

Este contrato estabelece o relacionamento genérico entre um gerenciador e o interceptador das mensagens, o qual é estendido para cada tipo de método de reflexão através de subcontratos.

CONTRACT ReflectionServices

MetaObjectManager **supports [**

reflectedObjects : **Set**(<Object,MetaObject>).

unreflect (anObject: Object) ⇒ Δ reflectedObjects; { !∃ metaObjectList / < anObject, metaObjectList> ∈ reflectedObjects }.

registerReflectionOf (anObject: Object; aMetaObject: MetaObject) ⇒ Δ reflectedObjects;

]

Interceptor **supports [**

component: Object.

manager: MetaObjectManager.

new(selectors: Set(Symbol); object: Object; manger: MetaObjectManager) ⇒
returns reflect (selectors;object;manager)

reflect (aCollection: Set;anObject:Object; manager: MetaObjectManager) ⇒
{◊ (Δ component; Δ manager)}

]

END CONTRACT

6.2.7.1.1 Mecanismo de Reflexão baseada nos métodos

Este contrato refina o contrato anterior para o caso do mecanismo de reflexão baseada nos métodos. O participante *MetaObjectManager* fornece serviços para refletir métodos. O subcontrato acrescenta o participante *Class*, o qual fornece os serviços para alterar a lista de métodos de uma classe, alguns desses métodos serão substituídos por métodos refletores, cujas responsabilidades são definidas pelo participante *ReflectedMethod*.

CONTRACT MethodReflectionServices

refines ReflectionServices (Interceptor = ReflectedMethod)

MetaObjectManager **supports** [

reflectMethod (selector: String, class: Class) ⇒
ReflectedMethod → new (oldMethod, selector, class, **current**);
{ ∃ method / <selector, method> ∈ class.compiledMethods ∧
type (method) = ReflectedMethod ∧
((< selector, oldMethod> ∈ **former** (class.compiledMethods) ∧ method.clientMethod = oldMethod) ∨
(< selector, oldReflectedMethod> ∈ **former** (class.compiledMethods) ∧
oldReflectedMethod.reflector ∧ oldReflectedMethod.clientMethod = method.clientMethod = oldMethod
)) }.

reflectMethodOn (selector: String; class: Class; metaObject: MetaObject) ⇒
reflectMethod (selector, class);
{∃ method : **type** (method) = ReflectedMethod : <selector, method> ∈ class.compiledMethods ∧
method.metaObject = metaObject}.

reflectCreationalMethodOf (class: Class) ⇒ { ∃ method /
<'new', method> ∈ class.compiledMethods ∧ **type** (method) = ReflectedMethod ∧
(method.virtual ∨ method.clientMethod = oldNewMethod ∧
<'new', oldNewMethod> ∈ **former** (class.compiledMethods))}.

]

Class **supports** [

compiledMethods: **Set** (< String, CompiledMethod>)

compiledMethodAt (selector) ⇒ **returns method**;
{<selector,method> ∈ compiledMethods}.

addMethod (selector, method) ⇒ Δ compiledMethods;
{ <selector, method> ∈ compiledMethods ∧
(!∃ otherMethod :otherMethod ≠ method : < selector, otherMethod > ∈ compiledMethods)
}.

removeMethod (selector) ⇒ Δ compiledMethods;
{<selector, method> ∉ compiledMethods}.

]

ReflectedMethod **supports** [

component: CompiledMethod.

reflect (selectors: Set; class:Class; manager: MetaObjectManager) ⇒

```
            returns r;
            { type(r): ReflectedMethod ;
              r = transferMethodFor(selector,class, manager);
              r.component = method; <selector, r> ∈ class. methods }.
        transferMethodFor(selector:Symbol,class:Class, manager:MetaObjectManager) ⇒
            returns Compiler→compile( selectorStringFor(selector, manager))
        selectorStringFor(selector, aManager) ⇒
            returns 'aManager→reflectMessage: selector args: arguments forObject: current'
    invariant
      Class.addMethod (aSelector, aMethod) ⇒ { ∀ <selector, method> :
        < selector, method > ∈ Class.compiledMethods : !∃ otherMethod,
        otherMethod ≠ method ∧ < selector, otherMethod > ∈ Class.compiledMethods}
      Class.removeMethod (aSelector) ⇒ { ∀ <selector, method> :
        < selector, method > ∈ Class.compiledMethods : !∃ otherMethod,
        otherMethod ≠ method ∧ < selector, otherMethod > ∈ Class.compiledMethods}
END CONTRACT
```

6.2.7.1.2 Mecanismo de Reflexão baseada nos objetos

Este contrato refina o contrato *ReflectionServices* para o caso do mecanismo de reflexão baseada nos objetos. O participante *MetaObjectManager* fornece serviços para refletir objetos. O subcontrato acrescenta o participante *Object*, o qual fornece os serviços para trocar a referência entre o objeto refletido e o *wrapper*, representado por *WrapperInterceptor*

```
CONTRACT ObjectReflectionServices

  refines ReflectionServices ( Interceptor = WrapperInterceptor)
  MetaObjectManager supports [
    wrapperReflect(selectors: Set(Symbol), object: Object, metaObject: MetaObject) ⇒
        r= WrapperInterceptor→ new (selectors, objects, current)
        r → reflect (selectors: Set; object:Object; manager: MetaObjectManager) ⇒
        { <r metaObject> ∈ reflectedObjects }
    wrapperUnreflect(wrapper: WrapperInterceptor) ⇒ ΔreflectedObjects;
        wrapper become(wrapper → component);
        { ∃ metaObject : <wrapper→component metaObject> ∉ reflectedObjects }
  Object supports [
      become(object:Object, wrapper:Object) ⇒
          { ref(object)= ref(wrapper) ∧ ref(wrapper) = former(ref(object))}
  ]
  WrapperInterceptor supports [
    component: Obejct.
```

```
    methods: Set(Symbol)

    reflect (selectors: Set; object:Object; manager: MetaObjectManager) ⇒ Δ methods
                current become(object)
                returns current;
                { methods = selectors}.

]
END CONTRACT
```

6.2.7.2 Ativação de meta-objetos

A ativação de meta-objetos envolve quatro participantes: o interceptador da mensagem que ativa o gerenciador, o gerenciador que decide quais meta-objetos devem ser ativados (se existe algum), os meta-objetos a serem ativados e a mensagem refletida, a qual, eventualmente receberá a mensagem *send* para assim executar o método original.

```
CONTRACT MetaObjectActivation

  Interceptor supports [

    manager: MetaObjectManager;

    "O seletor interceptado transfere o contorle para o manager"

    intercepted selector (args: List(Object)) ⇒

                manager → reflectMessage (selector, args, current)

  ]

  MetaObjectManager supports [

    reflectedObjects: Set ( < Object, Set (MetaObject) > ).

    reflecting: Boolean.

    isReflected (anObject: Object) ⇒

                        returns ( ∃ metaObjectList : <anObject, metaObjectList> ∈ reflectedObjects).

    reflectMessage (selector:Symbol, args:Collection, object:Object) ⇒
                if findMetaObjectFor (object) = ∅ ∨ ¬reflecting

                        then callBackMethod (selector, args, object)

                        else activateMetaObjects (findMetaObjectFor (object); reflectedMessage);

                        { type (reflectedMessage) = ReflectedMessage}.

    callBackMethod (selector:Symbol, args:Collection, object:Object) ⇒
                                object → perform(selector, args)

    startReflect () ⇒ Δ reflecting; { reflecting = true }.

    stopReflect () ⇒ Δ reflecting; { reflecting = false}.

    findMetaObjectsFor (object: Object) ⇒ returns listMetaObjects;
          { <object, listMetaObjects> ∈ reflectedObjects ∨ listMetaObjects = ∅}.

    activateMetaObjects (metaObjects: Set (MetaObject); aMessage: ReflectedMessage) ⇒
          {◊ ⟨ ∀ mo : mo ∈ metaObjects : mo → handleMessage (aMessage, current) ⟩ }.
```

```
]
MetaObjects Set(MetaObject) where each MetaObject supports [
    object: Object.
    setObject (anObject: Object) ⇒ Δ object; { object = anObject }.
    handleMessage(msg: ReflectedMessage, manager:MetaObjectManager) ⇒ { ◊ (msg → send)}
]
ReflectedMessage supports [
    new (receiver:Object, sender: Object; selector:Symbol, arguments: Set(Object)).
    execute () ⇒ receiver → perform(selector, arguments)
    send () ⇒ returns execute.
]
END CONTRACT
```

6.2.8 Definição de classes e conformação de contratos

Class MetaObjectManager

conforms to MetaObjectManager **in** MethodReflectionServices

MetaObjectManager **supports**[

atributes

ReflectedObjects: **Set(**Object**)**
role: *Mantém a associação entre objetos e meta-objetos*

base methods

reflectClass:(Class)on:(MetaOject)
role: *Trata a reflexão de métodos de classe sobre uma instância da classe de meta-objeto especificada.*

reflectInstancesOf:(Class) on:(MetaObject)
role:*Trata a reflexão de métodos de instância sobre uma instância da classe de meta-objeto especificada.*

reflectClass:(Class) upon:(MetaObject)
role: *Trata a reflexão de métodos de classe sobre um meta-objeto especificado.*

reflectInstancesOf:(Class) upon:(MetaObject)
role: *Trata a reflexão de métodos de instância sobre um meta-objeto especificado.*

reflectEachInstanceOf:(Class) on:(MetaObject)
role: *Trata a associação de cada instância de uma dada classe com instâncias diferentes da classe de meta-objeto especificada. Reflite todos os método de instância.*

reflectObject:(Object)on:(MetaObject)
role: *Trata a associação de um objeto com um dado meta-objeto*

reflectMethod:(String)of:(Class) on:(MetaObject)
role: *Reflite um dado método de uma dada classe sobre uma dado meta-objeto*

requires subclass to support

registerReflectionOf (anObject: Object, aMetaObject: MetaObject) ⇒ Δ reflectedObjects;
{ (∃ metaObjectList : <anObject, metaObjectList> ∈ reflectedObjects
aMetaObject ∈ metaObjectList) }.

conforms to MetaObjectManager **in** MetaObjectActivation

MetaObjectManager **supports**[

base methods

callBackMethod: (String) args:(Collection) :forObject:(Object)
role: *Executa o método original quando o gerenciador não está ativando meta-objetos ou não foram encontrados meta-objetos associados com o objeto base.*

template methods

reflectMessage:(String) args:(Collection) :forObject:(Object)
role: *Implementa o algoritmo genérico para ativação de meta-objetos quando uma mensagem é refletida. Poderia ser edefinido por razões de eficiência.*

subclass might redefine

canReflect
role: *Predicado. Retorna verdadeiro se o manager está activando meta-objetos. Este é o comportamento por falta.*

defaultMetaMessageClass
role: *Retorna a subclasse de ReflectedMessage utilizada pelo gerenciador para reificar mensagens refletidas. Este hook habilita a especificação de tipos alternativos de mensagens refletidos que poderiam resultar necessarios para aplicacões específicas*

stopReflect
role: *Inibe a ativação de meta-objetos pelo manager. As mensagens refletidas são direcionadas para o objeto original*

startReflect
role: *Ativa a ativação de meta-objetos pelo manager. As meesagens refletidas são direcionadas para o objeto original*

requires subclass to support

findMetaObjectsFor (object: Object)
role: *Deve retornar aqueles meta-objetos associados com o receptor da mensagem. Se não existem meta-objetos deve retornar nil.*

activateMetaObjects (metaObjects: **Set** (MetaObject), aMessage: ReflectedMessage)
as activateMetaObject: (Object) WithMessage (ReflectedMessage)
role: *Deve implementar a estratégia de transferencia de controle para os meta-objetos*

]

End Class MetaObjectManager
Class MetaObjectClass

conforms to MetaObject **in** MetaObjectActivation

MetaObject **supports**[

atributes

object: Object
role: *Mantém a referência ao objeto associado com o objeto.*

base methods

object
role: *Retorna o object associaso com o objeto. Por falta presume um único metaobjeto associado*

object:(Object)
role: *Set the object associated with the meta-object*

reflectedClassesAllowed
role: *Retorna as classes e métodos válidos que podem ser refletidas sobre esta classe de meta-objeto*

initializeWith: (Block)
role: *Especifica um bloco para incialização do meta-objeto*

requires subclass to support:

handleMessage:(ReflectedMessage) manager:(MetaObjectManager)
role: *Chamado por um manager para tratar uma mensagem refletida por um objeto base associado ao meta-objeto receptor. Deve ser implementado por subclasses.*

].
End Class MetaObjectClass

Class MessageInterceptor

conforms to Interceptor **in** MethodReflectionServices

MessageInterceptor **supports:[**

atributes

component: Object
role: *Objeto ou método interceptado*

manager: MetaObjectManager
role: *Manager sobre o qual as mensagems interceptadas são refletidas*

base methods

new(selectors: Set(Symbol); object: Object; manger: MetaObjectManager)
as onSelectors:(Collection) in: (Object) reflectingOn: (MetaObjectManager)
role: Interface para a reflexão de métodos de uma dada classe ou objeto sobre um dado gerenciador

manager:aMetaObjectManager
role: *Seta o gerenciador ao qual serão transferidas as mensagems interceptadas para o seu tratamento*

component:(Object or CompiledMethod)
role: *Seta o objeto ou método refletido*

requires subclass to support:

reflect (aCollection: Set;anObject:Object; manager: MetaObjectManager) ⇒ **{**◊ **(**Δ component, Δ manager)**}**
role: Deve implementar o algoritmo específico para refletir um dado conjunto de métodos de uma dada classe sobre um gerenciador

]
End Class MessageInterceptor

Class WrapperInterceptor

inherits from MessageInterceptor;

conforms to Interceptor **in** MetaObjectActivation

WrapperInterceptor **supports [**

atributes

methods: Set(Symbol)
role: *Métodos a serem interceptados*

base methods

intercepted selector (args: **List**(Object)) **as** doesNotUnderstand(aMessage: Message)
role:*transfere as mensagens para o manager se a mensagem deve ser refetida, se não tansfere o controle ao objeto*

reflect (aCollection: Set;anObject:Object; manager: MetaObjectManager) **as**
reflect: aCollection of: Object manager: aMetaObjectManager
***role**: Implementa o algoritmo para refletir um dado conjunto de métodos de uma dada insância sobre um dado gerenciador*

]

End Class WrapperInterceptor

Class ReflectedMethod

inherits from MessageInterceptor;

conforms to Interceptor **in** MetaObjectActivation

ReflectedMethod **supports [**

base methods

intercepted selector (args: **List**(Object))
role:*transfere as mensagens para o manager*

reflect (aCollection: Set;anObject:Object; manager: MetaObjectManager) **as**
reflect: aCollection of: Object manager: aMetaObjectManager
role: *Implementa o algoritmo para refletir um dado conjunto de métodos de uma dada classe sobre um gerenciador, substituindo os métodos compilados por instâncias próprias*

]

End Class ReflectedMethod

Class ReflectedMessage

conforms to ReflectedMessage **in** MetaObjectActivation

attributes

receiver: Object
role: *Receptor da mensagem*

sender:Object
role: *Originador da mensagem*

selector: Symbol
role: *Nome do método chamado*

arguments: Set(Object)
role: *Argumentos da mensagem*

value
role: *Mantém o valor resultante da primeira avaliação da mensagem. Este método pode ser redefinido para obrigar o reenvio da mensagem*

base methods

execute
role: *Executa o método original refletido*

senderObject
role: *Retorna o objeto originador da mensagem*

receiverObject
role:*Retorna o objeto receptor da mensagem*

senderClass
role: *Retorna a classe do objeto originador da mensagem. Se o originadors foi uma classe então retorna sua metaclasse.*

receiverClass
role: *Retorna a classe do objeto receptor da mensagem. Se o originadors foi uma classe então retorna sua metaclasse.*

homeReceiverClass
role:*Retorna a classe na qual a implementação do método foi encontrada na hierarquía de herança. Se o originador foi uma classe então retorna sua metaclasse.*

subclasses might redefine:

send
role: *Executa o método original refletido. Por falta só executa uma vez o método. Sucessivos envios da mensagem retornam o valor da primeira avaliação*

value
role:*Retorna o valor resultante da primeira avaliação da mensagem. Este método pode ser redefinido para obrigar o reenvio da mensagem.*

]
End Class ReflectedMessage

6.3 *LuthierBooks*: O sub-framework de representação

O sub-framework de representação, *LuthierBooks*, é baseado na implementação de gerência de hiperdocumentos do modelo PROMETO [ORT 95], especialmente projetado para suportar ambientes de desenvolvimento de software.

O modelo compõe-se de três classes de componentes: Entidades (representam elementos simples), Contextos (representam as composições de componentes) e Elos (modelam as associações entre componentes). A classe de um contexto determina as classes de instâncias que o contexto relaciona. Uma instância dessa classe de contexto, só admite como integrantes, instâncias das classes que tem sido definidas para sua classe.

6.3.1 Estrutura geral

A classe abstrata *Component* implementa o comportamento de objetos armazenados em um repositório de objetos. *Component* não fornece uma implementação específica deste armazenamento, más

somente fornece a interface para acessar a objetos que podem ser armazenados. Basicamente estabelece que cada componente possuirá um conjunto de atributos, um dos quais será utilizado como identificador do componente. Tanto o conjunto de atributos quanto o identificador são dependentes dos tipos específicos de componentes que sejam definidos. Os atributos são implementados como variáveis de instância dos distintos tipos de componentes.

Um elo é um tipo de componente que sempre vincula outros dois componentes. Um elo representa uma entidade associativa, no sentido de que somente pode existir quando existam os componentes que associa.

A classe *HypertextSystem* fornece a interface para interagir com um hiperdocumento armazenado em um banco de dados conceitual, tanto para armazenar como para acessar componentes. Também fornece a funcionalidade para criação e navegação de elos, bem como mantém informação das diversas visões que possui um componente. Fornece serviços tais como determinar todos os possíveis destinos de um elo de um tipo específico (uma vez estabelecida a origem), ou estabelecer com que tipo de visão deve um componente ser visualizado em um contexto determinado, por exemplo.

Os componentes de um hiperdocumento são acessados através de objetos intermediários, ou *proxies*, implementados pela classe *LuthierProxy*. Um *proxy* mantém a referência ao componente real, informação resumida acerca de seu conteúdo, como por exemplo um texto, e a informação necessária para recuperá-lo do armazenamento externo. *LuthierProxy* implementa um mecanismo *lazy* de recuperação do seu componente, o qual só é recuperado quando informação específica, além da resumida contida no próprio proxy é necessária. Assim, a recuperação de um contexto só envolve a leitura de um objeto e os proxies componentes, permitindo aumentar a eficiência no tratamento da informação.

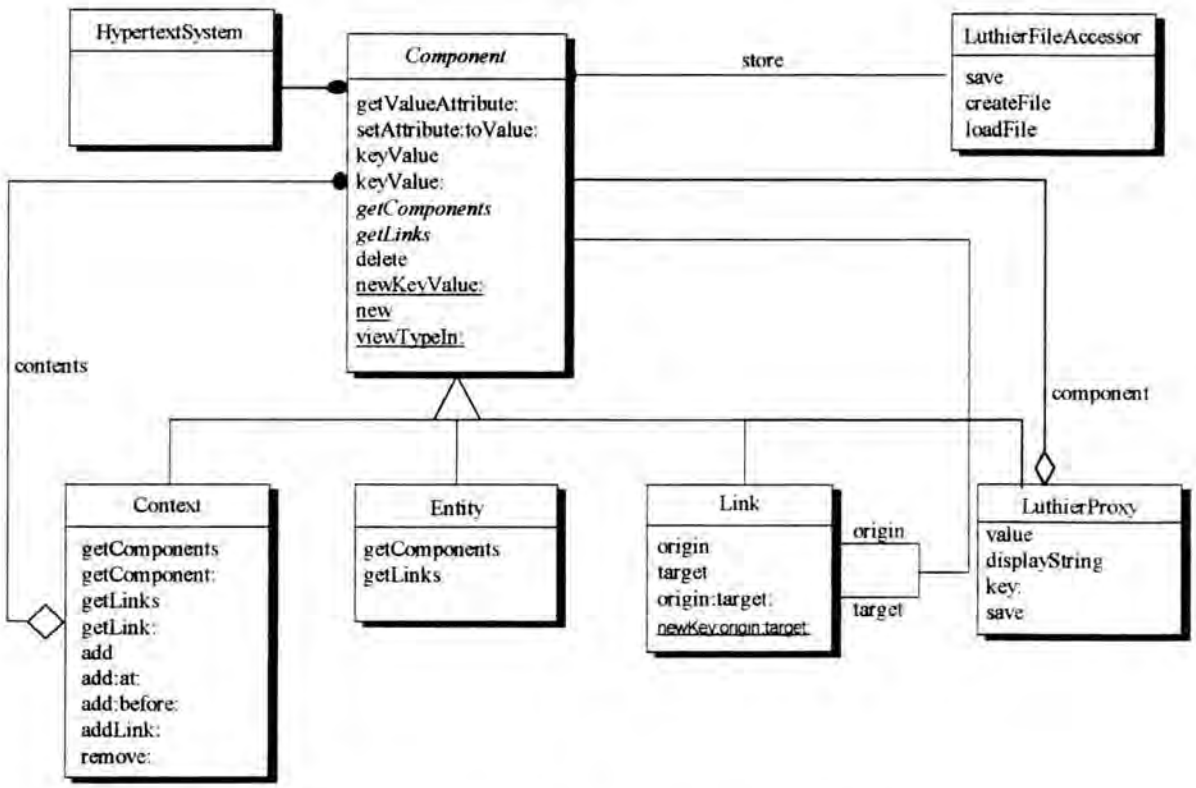


FIGURA 6.5- Modelo de objetos de LuthierBooks

O método de armazenamento é encapsulado pela classe *LuthierAccessor*, a qual é a encarregada de realizar a manipulação do meio de armazenamento, seja um arquivo ou um banco de dados. Isto permite tornar independentes os *proxies* do meio de armazenamento utilizado.

6.3.2 Definição de objetos simples

Objetos simples são aqueles componentes que não são formados por outros componentes. Objetos deste tipo devem ser definidos como subclasses da classe *Entity*, definindo todas as variáveis de instância e comportamento associado correspondente com o tipo de objeto que representam.

Exemplos de Instanciação:

Um exemplo de subclasse concreta é a classe *LuthierMethod*, que representa um método de uma classe do framework que esta sendo analisado. Alguns dos atributos acrescentados pela classe *LuthierMethod* são o nome do método, o tipo (instância o classe), a classificação (abstrato, *template*, *hook* ou base), os parâmetros que recebe, se é o não redefinido na hierarquia, o numero de vezes que foi invocado, a coleção de mensagens enviadas, entre outros.
A funcionalidade mais importante definida por esta classe é a necessária para acessar estes atributos. Neste casso, a classe não necessita redefinir o comportamento herdado.

```
Entity subclass: #LuthierMethod
        instanceVariableNames: 'name methodType timesCalled methodKind redefined
                                messages father applications parameters '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Luthier Hypertext Layer'
```

6.3.3 Definição de nodos agregados

Os contextos são o elemento de estruturação fundamental do modelo de hipertexto, já que suportam a composição recursiva de componentes. Através dos contextos é possivel representar como nodos agregados, objetos cuja semântica de associação pode representar tanto o relacionamento de composição como de conjunto. Diferentes contextos podem definir diferentes associações entre subconjuntos de um mesmo conjunto de objetos, de acordo a diferentes critérios de associação.

Esta capacidade permite reduzir de forma considerável a complexidade de uma rede formada exclusivamente por nodos simples e elos, além de permitir utilizar elos como mecanismo de associação binária de elementos relacionados e navegação sobre a rede.

A classe *Context* fornece a funcionalidade especifica para acessar e mudar o conjunto de componentes que formam um contexto dado.

Exemplos de Instanciação:

A classe *LuthierClass* representa as classes que compõem um framework que está sendo analisado. O conteúdo de um contexto deste tipo são os métodos da classe (instâncias de *LuthierMethod*) e as subclasses definidas. Exemplos de atributos definidos por esta classe são o nome, o pai (superclasse) e as aplicações nas quais a classe é utilizada.
O método **add:** especializa o comportamento herdado, para colocar o receptor como pai do objeto acrescentado.

```
Context subclass: #LuthierClass
    instanceVariableNames: ' name father applications '
    classVariableNames: ''
    poolDictionaries: ''
```

```
    category: 'Luthier Hypertext Layer'!

add:anEntity
    super add: anEntity
    anEntity father:self.
    ^anEntity.
```

O método **topComponent** retorna o componente mais abstrato da hierarquia (isto é, um LuthierClass)

```
topComponent
    ^father notNil ifTrue:[ father topComponent] ifFalse:[ self]
```

6.3.4 Criação de elos

A classe *Link* implementa a funcionalidade necessária para estabelecer um relacionamento entre dois componentes; corresponde com a noção de elo de um modelo de hipertexto, permitindo às *Entities* e *Contexts* ter o papel de nodos neste modelo de hipertexto.

Exemplos de Instanciação:

A classe *LuthierMessage* é utilizada para modelar os relacionamentos cliente-servidor que existem entre os métodos. Exemplos dos atributos que possui são a quantidade de vezes que a mensagem foi enviada, as classes das instâncias que enviaram e receberam a mensagem, e outros atributos úteis para implementar a funcionalidade fornecida pelo ambiente.

```
LinkComponent subclass: #LuthierMessage
instanceVariableNames: 'timesActivated realReceiverContext realSenderContext
implementorContext senderInstance receiverInstance messageCategory initialThread finalThread
applications listActivated '
classVariableNames: ''
poolDictionaries: ''
category: 'Luthier Hypertext Layer'
```

Um método especializado pela classe é o método *origin:target:*. Este método, alem de instanciar as variáveis *origin* e *target*, tem que acrescentar ao método de origem uma nova mensagem enviada.

```
origin:aNode target:anotherNode
        super origin:aNode target:anotherNode.
        aNode message: self.
        ^self.
```

6.3.5 Gerenciamento de hiperdocumentos e navegação

A classe *HiperTextSystem* fornece uma interface geral para o gerenciamento de hiperdocumentos contidos em uma base de dados conceitual. Esta classe é projetada para ter só uma instância.

A interface oferecida por *HyperTextSystem* consiste, essencialmente, de métodos para:

- criação de componentes acessíveis através de uma chave.
- recuperação de componentes através de uma chave
- recuperação de elos desde ou para um componente dado

Exemplos de Instanciação:

O método seguinte realiza a criação de um componente que representa um método de uma classe, representado por instâncias da classe de componente *LuthierMethod*. O método recupera um componente com o valor de chave dado, através da mensagem ***entityWithKeyValue:***. Caso não exista já um contexto com essa chave, o contexto é criado através da mensagem ***addNodeType:keyValue:***. A mensagem ***htSystem*** retorna a instância corrente do sistema de hipertexto.

```
addMethod: aSelector
        "Creates the participant if it doesn't exist yet"
        | meth name |
        name := self objectName.
        (meth := HyperTextSystem htSystem entityWithKeyValue: aSelector '-' name) isNil
                ifTrue:
                        ["Creates the method component"
                        meth := HyperTextSystem htSystem addNodeType: #LuthierMethod
                                                keyValue: aSelector '-' name.
                        meth := self contextParticipant add: meth at: 1 @ 1]
                ifFalse:[ meth called].
        ^meth
```

A criação de um elo novo entre dois componentes é realizada de forma análoga. Por exemplo a criação de um elo de tipo *LuthierMessage* entre dois componentes de tipo LuthierMethod é realizada da forma mostrada abaixo

```
(link:=HyperTextSystem htSystem linkWithKeyValue: name) isNil
        ifTrue:[ HyperTextSystem htSystem createLinkFrom: aMethodComponent
                                to: anotherMethodComponent
                                linkType: #LuthierMessage
                                keyValue: name.
```

Para implementar funções de navegação, as seguintes funções são providas:

HyperTextSystem htSystem allLinkFrom: aComponent
retorna todos os elos cuja origem é o componente dado.

HyperTextSystem htSystem allLinkTo: aComponent
retorna todos os elos cujo destino é o componente dado

HyperTextSystem htSystem allLinkTypesFrom: aComponent
retorna todos os tipos de elos cuja origem é o componente dado

HyperTextSystem htSystem allTargetsFrom: aComponent **linkType:**aType
retorna todos os elos do tipo dado cujo destino é o componente dado

6.3.6 Tornando persistentes os componentes

Quando novos componentes são criados através do gerenciador de hiperdocumentos, o gerenciador associa automaticamente uma instância de *LuthierProxy*, a qual mantém a informação necessária para recuperar o componente do armazenamento externo.

Em circunstâncias nas quais um componente de representação é criado diretamente pela aplicação, pode acrescentar-se a capacidade de persistência associando-lhe uma instância de *LuthierProxy*. Um *proxy* encapsula a informação relativa ao meio de armazenamento e a chave do seu componente, bem como, informação elementar, como por exemplo um texto descritivo, que pode ser utilizada sem necessidade de recuperar o componente do armazenamento externo.

Para tornar persistente um componente é necessário criar uma instância de *LuthierProxy* sobre esse componente e enviar ao *proxy* a mensagem *save*. Esta mensagens produzirá que o componente seja armazenado no meio definido pelo gerenciador de armazenamento *LuthierAccessor*.

Exemplos de Instanciação:

O exemplo a seguir mostra a criação de um livro persistente de documentação em uma biblioteca. *LuthierBook* é a classe de contexto que contém a informação relativa ao livro. O método *defaultStorer* retorna a instância corrente de *LuthierAccessor*, a qual é instruída para criar um novo arquivo para o caminho definido pelo nome do livro. Para tornar persistente o livro, é criada uma instância de *LuthierProxy* sobre o livro e o gerente de armazenamento é instruído para inicializar o arquivo com o proxy que referencia o livro. A mensagem *storer* informa ao *proxy* seu atual meio de armazenamento, o qual é utilizado, posteriormente, para setar o *accessor* corrente (o nome do arquivo neste caso). Finalmente, o proxy é agregado ao contexto que representa a biblioteca.

```
newBook: aTitle
        |book fileIndex proxy path |
        book := LuthierBook title:aTitle.
        path:= BookDirectory, book fileId.
        Books add:path.
        fileIndex:= LuthierProxy defaultStorer addFile: path.
        proxy:= LuthierProxy on: book.
        proxy storer: fileIndex.
        LuthierProxy defaultStorer startFileFor: proxy.
        ^self add: proxy
```

6.3.7 Especificação de contratos

6.3.7.1 Relacionamento geral entre contextos e componentes

Este contrato especifica o relacionamento entre um objeto composto (*Context*) e os seus componentes (*Component* e *Link*). O Context terá uma coleção de Componentes e uma de Elos, além de uma lista, na qual os componentes serão acessíveis pela sua chave.

O invariante do contrato estabelece que não podem existir dois objetos com a mesma chave, e esta condição deve ser preservada pelas funções *add*, *add-at*, *add-before*, e *addLink* do participante *Component*. Uma condição adicional que deve ser respeitada pela função *add-before* é que o elemento inserido fique antes do elemento provido como argumento.

```
CONTRACT Context-Component-Link

Context supports [
        contents: Set (Component).
        links: Set (Link).
        keys: Set (<KeyClass,Component>).
```

```
    add (aComponent: Component) ⇒ Δ contents; Δ keys;
        {aComponent ∈ contents; <aComponent.key,aComponent> ∈ keys ∧
          ( !∃ otherComponent : otherComponent ≠ aComponent :
              <aComponent.key, otherComponent > ∈ keys)}.
    addAt (aComponent: Component; aPoint: Point) ⇒ add (aComponent).
    addBefore (component: Component, anotherComponent: Component) ⇒ Δ contents; Δ keys;
        {component ∈ contents ∧ anotherComponent ∈ contents ∧
            component ≺ anotherComponent ∧
            ( !∃ thirdComponent : thirdComponent ∈ contents :
                component ≺ thirdComponent ≺ anotherComponent).
            <component.key,component> ∈ keys ∧
            ( !∃ otherComponent : otherComponent ≠ aComponent :
                <component.key, otherComponent > ∈ keys)}.
    addLink (aLink: LinkComponent) ⇒ Δ links;
        {aLink ∈ links; <aLink.key,aLink> ∈ keys ∧
            !∃ otherLink, otherLink ≠ aLink, <aLink.key, otherLink> ∈ keys}.
    getLinks ⇒ returns links.
    getNodes ⇒ returns contents.
    getComponent (key: KeyClass) ⇒ returns aComponent; {<key, aComponent> ∈ keys}.
    getLink (key: KeyClass) ⇒ returns aLink; {<key, aLink> ∈ keys}.
    keys ⇒ returns keys.
    remove (aComponent: Component) ⇒ Δ contents; Δ links;
        {aComponent ∉ contents; <aComponent.key,aComponent> ∉ keys}
].
Components: Set(Component) where each Component supports [
    key: KeyClass.
Links: Set(Link) where each Link supports [
    key: KeyClass.
    origin: Component.
    target: Component.
    new(key: KeyClass, origin:Component, target:Component) ⇒
        setOriginTarget:(aComponent: Component; anotherComponent: Component).
        { current. key = key }
    setOriginTarget:(aComponent: Component; anotherComponent: Component) ⇒ Δ origin; Δ target
```

{ origin = aComponent ∧ target = anotherComponent}

].

invariant

Context.add (component) ⇒ ⟨∀c : c ∈ Components : (∀<aKey, aComponent> : <aKey, aComponent> ∈ c.keys : !∃ otherComponent, otherComponent ≠ aComponent ∧ <aKey, otherComponent> ∈ c.keys) ⟩.

Context.addAt (component, point) ⇒ ⟨∀c : c ∈ Components : (∀<aKey, aComponent> : <aKey, aComponent> ∈ c.keys : !∃ otherComponent, otherComponent ≠ aComponent ∧ <aKey, otherComponent> ∈ c.keys) ⟩.

Context.addBefore (component, beforeComponent) ⇒ ⟨∀c : c ∈ Components : (∀<aKey, aComponent> : <aKey, aComponent> ∈ c.keys : !∃ otherComponent, otherComponent ≠ aComponent ∧ <aKey, otherComponent> ∈ c.keys) ⟩.

Context.addLink (link) ⇒ ⟨∀c : c ∈ Components : (∀<aKey, aComponent> : <aKey, aComponent> ∈ c.keys : !∃ otherComponent, otherComponent ≠ aComponent ∧ <aKey, otherComponent> ∈ c.keys) ⟩.

END CONTRACT

CONTRACT HypertextSystem - Component - Link

includes Context-Component-Link

HypertextSystem **supports** [

nodes: **Set** (< KeyClass, Component >).

links: **Set** (< KeyClass, Link >).

allLinksFrom (aComponent: Component) ⇒ **returns** l;
{ l= **Set**(link | link ∈ links.values ∧ link.origin = aComponent) }.

createLink (origin, target: Component; linkType: LinkClass; keyValue: KeyClass) ⇒
Δ links; **returns** newLink;
{**type** (newLink) = linkType; newLink.key = keyValue; newLink.origin = origin; newLink.target = target; newLink ∈ links}.

addNode (nodeType: ComponentClass; keyValue: KeyClass) ⇒
Δ nodes; **returns** newNode;
{ **type** (newNode) = ComponentClass; newNode.key = keyValue;
< keyValue, aComponent> ∈ nodes ∧ (!∃ otherComponent :
otherComponent ≠ aComponent : < keyValue, otherComponent > ∈ nodes)}.

entityWithKeyValue (keyValue: KeyClass) ⇒ **returns** node;
{ < keyValue, node > ∈ nodes }.

linkWithKeyValue (keyValue: KeyClass) ⇒ **returns** link;
{ < keyValue, link > ∈ link }.

removeLink (alink: Link) ⇒ Δ links; **returns** alink; { alink ∉ links }.

removeComponent (aComponent: Component) ⇒ Δ nodes; **returns** aComponent;
{ aComponent ∉ nodes }.

].

Components: **Set**(Component) **where each** Component **supports** [

].

Links: **Set**(Link) **where each** Link **supports** [

].

invariant

HypertextSystem.addNode (nodeType, keyValue) ⇒ ⟨∀ <aKey, aComponent> :
<aKey, aComponent> ∈ HypertextSystem.nodes :
!∃ otherComponent, otherComponent ≠ aComponent ∧
<aKey, otherComponent> ∈ HypertextSystem.nodes ⟩.

END CONTRACT

6.3.7.2 Persistência de componentes

Este contrato especifica o protocolo entre componentes e *proxies*. A funcionalidade básica de um *proxy* é propagar as mensagens que recebe ao seu componente, o qual é solicitado através da mensagens *value*. Se o componente ainda não foi recuperado, então, o recupera e propaga a mensagem. O contrato inclui as colaborações definidas pelos contratos *Model-Dependent*, pois um *proxy* depende do seu componente para conhecer mudanças na chave, e o contrato *Context-Component-Link*, pois o *proxy* também representa contextos e elos.

CONTRACT Proxy-Component-Accessor

Proxy **supports** [

contents: Component.

displayString: **String**.

key: KeyClass.

storer: **String**.

on(component: Component) ⇒ **return** p
{ **type**(p) = Proxy ∧ p.component = component}

value () ⇒ **if** (contents = ∅ **&** key ≠ ∅) **then** [refresh; updateDisplayString];
returns contents.

allNodes () ⇒ **returns** value → allNodes.

getNodes () ⇒ **returns** (value → getNodes).

isKindOf (class: Class) ⇒ **returns** (**type** (**current**) = class ∨ value → isKindOf (class)).

keyValue () ⇒ **returns** key.

displayString () ⇒ **returns** displayString;{ displayString ≠ ∅ }.

```
    setDisplayString (newString:String) ⇒ Δ displayString; {displayString = newString}.

    updateDisplayString () ⇒ Δ displayString; { contents → keyValue ≠ ∅ ⇒
            displayString = contents → keyValue }.

    setContents (newContents: Component) ⇒ Δ contents;
            newContents → addDependent (current);
            updateDisplayString()
            { contents = newContents }.

    update () ⇒ updateDisplayString.

    save () ⇒ updateDisplayString.

    doesNotUnderstand (message: Message) ⇒ returns value → message.

    dumpContents () ⇒ remove; updatePrintString; { contents = ∅ }.

]
Component supports [

    key: KeyClass.

    isKindOf (class:Class) ⇒ returns ( type (current) = class ).

    keyValue () ⇒ returns key.

    embeddedProxies () ⇒ returns proxies; { ∀ p : p ∈ proxies : p ∈ contents.contents ∧
            type (p) = Proxy }.

]
includes Model-Dependent ( Model = Component; Dependent = Proxy).

includes Context-Component-Link ( Context = Component; Component = Proxy;
Link = Proxy).

includes Proxy-Accessor
```

END CONTRACT

6.3.7.3 Armazenamento de componentes

Este contrato especifica a colaboração entre um *proxy* e o gerenciador de armazenamento. O gerenciador mantém a referência à unidade de armazenamento corrente, representada genericamente pelo participante *Storer*. O gerenciador só armazena o conteúdo do *proxy*, e não o *proxy* mesmo, assim, um *proxy* só é salvado quando é contido por um outro *proxy*.

CONTRACT Proxy-Accessor

```
Proxy supports [

    storer: Storer.

    storer(aStorer)⇒ Δ storer { storer = aStorer}

    refresh () ⇒ Accessor → refresh (current).

    save () ⇒ Accessor → append (current).
```

]

Accessor **supports** [

currentWriteStorer: Storer

currentReadStorer: Storer

refresh (proxy: Proxy) ⇒ currentReadStorer(proxy.storer);
contents=currentReadStorer → **read** proxy.key;
{ proxy.contents = contents}

append(proxy: Proxy) ⇒ currentWriteStorer(proxy.storer);
currentStorer → **store** proxy→contents ;
{ currentWriteStorer = proxy.storer ∧ currentKey = proxy.key }

currentReadTarget (aProxy: Proxy) ⇒ { currentReadStorer = proxy.storer ∧ currentKey = proxy.key }

currentReadTarget (proxy.index);

]

Storer **supports** [

contents: **Set**(<Key, Object>)

read(aKey: Key) ⇒ {<aKey, anObject> ∈ contents>}
return(Object).

store(aKey: Key, anObject:Object) ⇒ { <aKey,anObject> ∈ contents}

]

END CONTRACT

6.3.8 Definição de classes e conformação de contratos

Class Component

conforms to Component **in** Context-Component-Link

Component **supports**[

atributes

keyVar: String
role: *Indica o nome da variável que representa a chave de acesso às entidades dessa classe.*

base methods

getValueAttribute: (String)
role: *Retorna o valor do atributo especificado.*

setAttribute: (String) toValue:(Object)
role: *Muda o valor do atributo especificado a um valor dado.*

keyValue
role: *Retorna o valor do atributo definido como chave para a classe.*

keyValue: (String)
role: *Muda o valor do atributo definido como chave para a classe a um valor dado.*

newKeyValue: (String)
role: *Cria uma nova instância do receptor, armazenando-a no repositório de objetos. O identificador para acessar o novo componente é fornecido como parâmetro. Retorna o novo componente.*

new
role: *Cria uma nova instância do receptor, armazenando-a no repositório de objetos. O identificador para acessar o novo componente é criado automaticamente, mas pode ser mudado.*

viewTypeIn:(Context)
role: *Retorna o tipo de visão que deve ser utilizado para representar uma instância do receptor em um contexto dado.*

subclasses might redefine

delete
role: *Elimina o receptor do repositório de objetos.*

requires subclasses to support

getNodes
role: *Retorna os elementos que compõe o receptor, caso que o componente seja complexo.*

getLinks
role: *Retorna os elos que compõe o receptor, caso que o componente esteja composto por elos.*

]

conforms to Component **in** Component-Proxy

Component **supports**[

embeddedProxies
role: *Retorna os proxies contidos entre seus componentes.*

]

End Class Component

Class Context

conforms to Context **in** Context-Component-Link

Context **supports**[

atributes

contents
role: *Mantém o conjunto de componentes (que não são elos) que formam parte do contexto.*

links
role: *Mantém o conjunto de elos que formam parte do contexto.*

keys
role: *Mantém o conjunto de identificadores dos componentes que contém.*

base methods

getNodes
role: *Retorna o conjunto vazio, dado que um objeto do tipo Entity não é composto por outros elementos.*

getLinks

role: *Retorna o conjunto vazio, dado que um objeto do tipo Entity não é composto por outros elementos.*

subclasses might redefine

add:(Component)
role: *Acrescenta um componente ao conteúdo do receptor.*

add: (Component)at:(Point)
role: *Acrescenta um componente ao conteúdo do receptor; o ponto de inserção é utilizado para informar às apresentações.*

add: (Component) Before: (Component)
role: *Acrescenta um componente ao conteúdo do receptor, antes de um componente dado.*

addLink:(Link)
role: *Acrescenta um elo ao conteúdo do receptor.*

getComponent: (String)
role: *Retorna o componente do contexto receptor cujo identificador é o valor dado como argumento.*

getLink: (String)
role: *Retorna o elo do contexto receptor cujo identificador é o valor dado como argumento.*

remove:(Component)
role: *Elimina um componente do conteúdo do contexto.*

End Class Context

Class Entity

inherits from Component

base methods

getNodes
role: *Retorna o conjunto vazio, dado que um objeto do tipo Entity não é composto por outros elementos.*

getLinks
role: *Retorna o conjunto vazio, dado que um objeto do tipo Entity não é composto por outros elementos.*

End Class Entity

Class Link

conforms to Link **in** Context-Component-Link

Link **supports:[**

attributes

origin
role: *Mantém o componente origem do relacionamento.*

target
role: *Mantém o componente destino do relacionamento.*

base methods

new(key: KeyClass, origin:Component, target:Component) **as**
newKey:(String)origin: (Component)target: (Component)

role: *Cria um novo elo, com o identificador, origem e destino dados.*

subclasses might redefine:

origin
role: *Retorna o componente origem do relacionamento.*

target
role: *Retorna o componente destino do relacionamento.*

setOriginTarget:(aComponent: Component; anotherComponent: Component) **as**

origin:(Component) target: (Component)
role: *Estabelece novos origem e destino para o relacionamento*

]

End Class Link

Class LuthierProxy

conforms to Proxy **in** Proxy-Component

LuthierProxy **supports [**

attributes

storer: Symbol

contents:Object.
role: *Mantém a referência ao componente do proxy.*

displayString: String.
role: *Mantém a informação básica do componente para ser mostrada evitando acessar o componente.*

key: String.
role: *Chave para acessar o componente no armazenamento persistente.*

base methods

getNodes
role: *Propaga o identificador de arquivo a seus componentes.*

dumpContents
role: *Elimina a referência ao seu componente.*

displayString
role: *Retorna a informação básica do componente. Habitualmente um texto a ser apresentado.*

updateDisplayString
role: *Atualiza a informação básica do componente..*

setContents **(**newContents: Component**) as**
contents:newContents
role: *Fixa o componente.*

doesNotUnderstand **(**message: Message**) as**
doesNotUnderstand: aMessage
role: *Propaga ao seu valor a mensagem aMessage*

subclasses maight redefine

value

role: *Retorna seu componente. Se o componente ainda não foi recuperado, o recupera através da chave*

]

conforms to Proxy **in** Proxy-Accessor

LuthierProxy **supports:[**

attributes

storer: String

base methods

refresh

role: *recupera seu componente do armazenamento persistente.*

save

role: *Salva o seu componente no armazenamento persistente.*

]

End Class LuthierProxy

6.4 *LuthierViews*: O Sub-Framework de Visualização

O sub-framework LuthierBooks é uma extensão do suporte gráfico provido pelo ambiente Smalltalk-80, o qual se baseia na composição hierárquica de objetos visuais [PAR 94].

O objetivo principal de projeto do sub-framework é permitir variar dinamicamente tanto distribuição espacial dos objetos visuais quanto a apresentação visual destes objetos e sua manipulação através do mouse.

6.4.1 Estrutura básica

A FIGURA 6.6 apresenta o diagrama de objetos das classes que compõem *LuthierViews*. A classe Model é destacada para denotar qualquer classe que possa ser um modelo das visões. A maior parte das classes são derivadas a partir das classes providas pelo framework MVC de Smalltalk, modificando, essencialmente, o mecanismo de tratamento de eventos de mouse e de atualização das apresentações. A classe referenciada como *LuthierEventHandlerView* na figura, é decomposta, na implementação, em três subclasses que correspondem com a divisão realizada por Smalltalk: visão que tem controlador associado, denominada no framework *LuthierEventHandlerCompositeView*, visão composta, denominada *LuthierEventHandlerComposite*, e visão simples, denominada *LuthierEventHandlerPart*. As três classes implementam a mesma interface relativa ao manejo de eventos, enquanto o comportamento restante é herdado das classes definidas no ambiente Smalltalk. Esta estrutura produz uma duplicação de alguns comportamentos, como por exemplo nas classes *LuthierStrategyView* e *LuthierStrategyComposite*, as quais acrescentam a estratégia de distribuição à funcionalidade definida por *LuthierEventHandler CompositeView e LuthierEventHandlerComposite*. Esta duplicação é justificada, pois evita a necessidade de reimplementar os mecanismos de dimensionamento e manipulação de subvisões que são fornecidos pelas classes do MVC.

Uma visualização construida com LuthierViews estará formada por um conjunto de objetos organizados em uma hierarquia duplamente conectada de visões e subvisões. A visão pai é a encarregada de definir a área que terão as visões filhas, de acordo com o protocolo estabelecido pelo MVC, descrito nos capítulos 2 e 3. Cada objeto tem associado um modelo que contém a informação a ser visualizada.

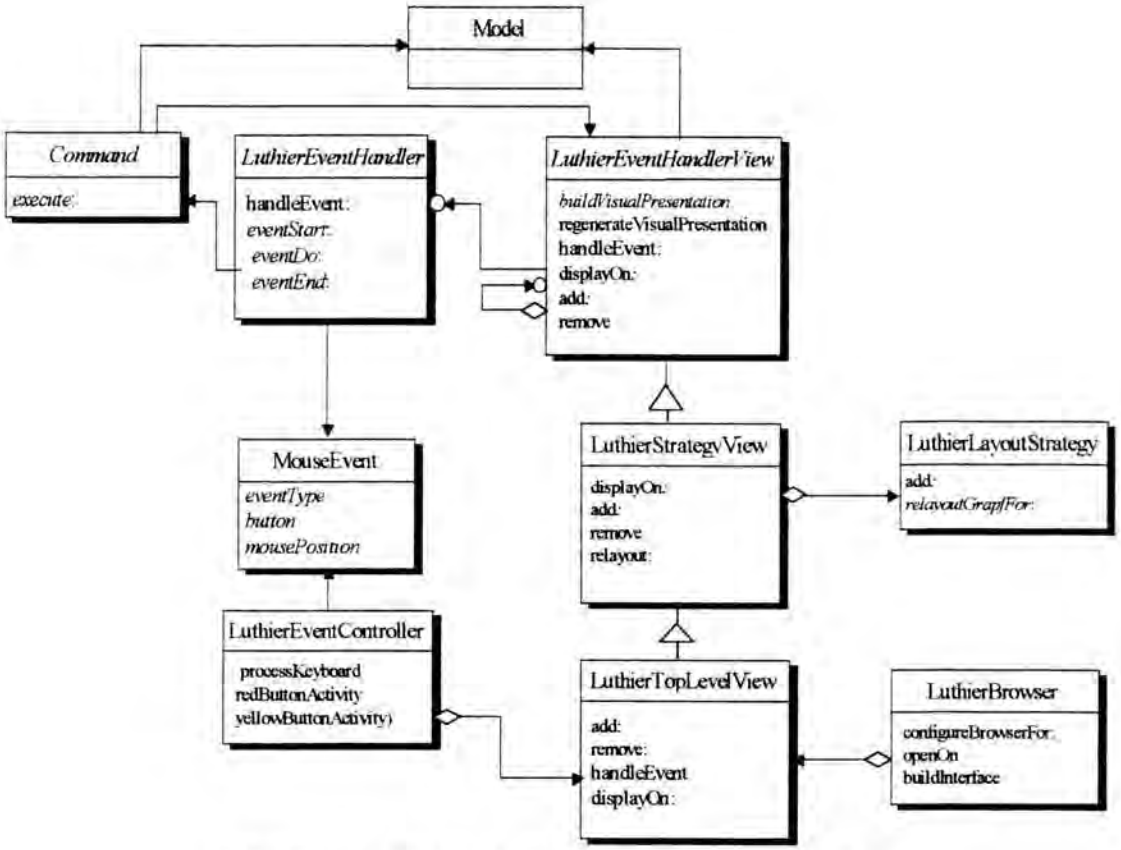


FIGURA 6.6- Modelo de Objetos Parcial de LuthierViews

O topo dessa hierarquia de instâncias deve ser uma instância de uma subclasse de *LuthierTopLevelView*, a qual tem associada uma instância de *LuthierEventController*, encarregada de converter o mecanismo de eventos de mouse e teclado padrão de Smalltalk, para o mecanismo de transferencia de eventos implementado por Luthier.

A subclasse *LuthierStrategyView*, provê o suporte para implementar visualizações abstratas que permitem variar dinamicamente a distribuição espacial da representação gráfica da informação a ser visualizada. Uma visão deste tipo é composta por uma *estratégia* [GAM 94] de distribuição (*LuthierLayoutStrategy*), a qual encapsula o algoritmo que distribui os objetos na área de visualização. A visão propaga as mensagens que recebe para agregar componentes (*add*) a sua estratégia associada, a qual define a posição onde o objeto será localizado. Com este suporte, diferentes algoritmos de distribuição de grafos podem ser providos, os quais podem ser trocados dinamicamente durante a execução de uma ferramenta.

A FIGURA 6.7 apresenta um cenário abstrato que descreve as interações entre as visões e as estratégias. Quando uma visão recebe seu modelo, cria sua estrutura de subvisões para representar a informação do modelo, através da mensagem *buildVisualPresentation*. A implementação deste método deverá criar instâncias de subvisões e acrescentá-las à visão composta através do método *addNode*. Este método solicita à estratégia associada a inserção da nova visão na posição que seja adequada. Em uma segunda interação a visão recebe a mensagem *displayOn:* para produzir na tela o desenho correspondente. A classe *LuthierTopLevelView* produz que todas as visões seja redistribuídas, caso exista alguma que tenha informado que mudou de tamanho enviando à estratégia correspondente a mensagem *relayoutGraphFor:* (este é um mecanismo de *lazy* de redistribuição que evita múltiplas redistribuições do mesmo elemento

quando existem mudanças nos tamanhos das visões), para depois propagar para as subvisões a mensagem para produzir o desenho.

Cada visão em *LuthierViews* tem associados um conjunto de manejadores de eventos de mouse (classe *LuthierEventHandler*), os quais implementam as manipulações típicas de mouse, como tratamento de *click*, arraste, etc. Estas manipulações podem ser dinamicamente associadas com qualquer visão, permitindo, deste modo, incorporar manipulação direta a qualquer visualização, sem necessidade de programar para cada visão particular as manipulações de mouse correspondentes.

Quando o usuário interage com o mouse sobre uma representação, o controlador associado com a visão de nível superior (*LuthierTopLevelView*) envia à visão a mensagem *handleEvent:*, com uma instância de *LuthierMouseEvent* como argumento. Esta instância contém toda a informação relativa ao evento.

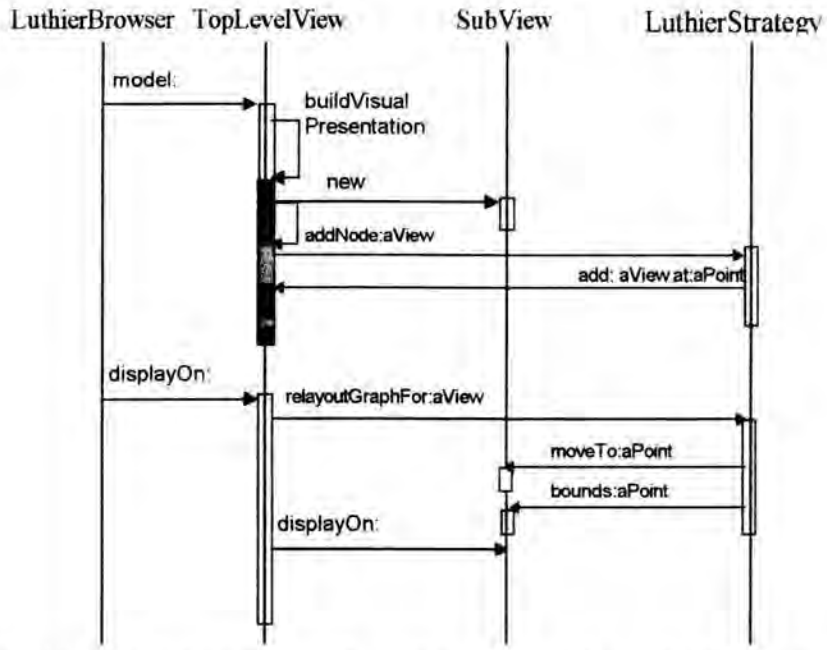


FIGURA 6.7- Fluxo de controle genérico entre visões, subvisões e estratégias

O evento é propagado desde o nível inferior da hierarquia (ou seja, as visões mais externas) entre os possíveis manejadores associados, até que algum de eles decida tratá-lo. Se nenhum decide tratá-lo, o evento é propagado para acima na hierarquia de visões, caso a visão pai esteja interessada no evento.

Se um manejador interessado no evento é achado, a visão transfere o controle para esse manejador, o qual implementa o tratamento de toda a manipulação. Por exemplo, o arraste de uma figura até que o botão é livrado. Terminada a manipulação, o manejador executa um *comando* associado, caso exista algum.

Um comando (classe *Command*) encapsula uma operação a ser aplicada tanto sobre a visualização (*seleção*, por exemplo) como sobre a representação dos dados. Os comandos podem ser dinamicamente associados com os manejadores de eventos, permitindo assim, mudar o comportamento da interface dinamicamente. Os comandos respondem à mensagem *execute* a qual deve ser implementada para cada operação específica.[1]

---

[1] Esta arquitetura de interfaces é baseada no modelo proposto em [SZE 88], no qual os manejadores de eventos são denominados *Recognizers*.

A FIGURA 6.8 apresenta um cenário abstrato da transferência de controle entre os diferentes objetos envolvidos no tratamento de eventos.

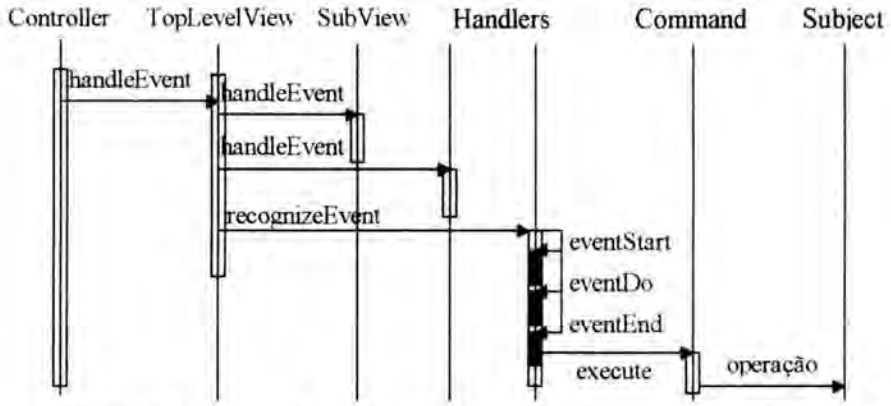


FIGURA 6.8- Fluxo abstrato de mensagens para o tratamento de eventos

A estrutura básica de uma visualização estará constituída por uma subclasse de *LuthierBrowser*, a qual fornece a interface abstrata seguinte:

- *openInterface*: Criação da interface externa da ferramenta encapsulando o protocolo específico de MVC para criar janelas.
- *configureBrowser(Context)*: criação da associação entre a visão de nível superior requerida, os abstratores e o contexto que contém a informação a ser visualizada.
- *openOn(Context)*: template para criação da estrutura do *browser*.

A interface externa da visualização, como por exemplo, a barra de menu e a distribuição de visões, pode ser realizada através do editor de interfaces provido pelo framework MVC.

A visualização final é construída quando os modelos das visualizações são especificados, conforme explicado na seção seguinte.

Exemplo de Instanciação

A classe *CollaborationBrowser* implementa um browser para uma visualização de grafo de fluxo de mensagens, cuja visão de nível superior é implementada pela classe *CollaborationView*. A visualização tem como modelo uma composição de abstratores (ver Criação e Composição de Abstratores) sobre o contexto que contém a informação a ser visualizada.

```
LuthierBrowser subclass: #CollaborationBrowser
        instanceVariableNames: ' abstractor topLevelView '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Luthier Browsers'

openOn: aContext
        self configureBrowserFor: aContext.
        self openInterface. "Abre a interface definida com o editor"

configureBrowserFor: aContext
        "Cria a estrtura básica da visualização
        abstractor := FrameworkAbstractor on: ( MessagePropertiesAbstractor on: aContext ).
        topLevelView:= CollaborationView new.
```

```
abstractor view: topLevelView.
subject := aContext addDependent: self.
topLevelView model: abstractor.
```

Um cliente abrirá uma instância da visualização através da mensagem

```
CollaborationBrowser new openOn: aContext.
```

6.4.2 Construção de visualizações compostas

No caso geral, uma visão é composta por subvisões, as quais constituem os *componentes* da visão composta. Para implementar a estrutura de uma visualização composta, é necessário criar instâncias das classes que implementam cada um dos componentes visuais que compõem a visão e adicioná-los à lista de componentes da visão composta.

O método *buildVisualPresentation*, deve ser implementado por cada visão particular para construir a estrutura de objetos visuais que será posteriormente desenhada na tela. Este método é automaticamente chamado quando a visão recebe seu modelo:

```
model: aLuthierView
        model == aLuthierView ifTrue:[^self].
        super model: aLuthierView.
        self buildVisualPresentation.
```

Uma visão em LuthierViews é construida, geralmente, através de um mecanismo que consiste em solicitar ao modelo da visão a informação a ser visualizada, criar as instâncias de subvisões que correspondem com cada tipo de dado informado pelo modelo, adicionar estas instâncias à estrutura de componentes e setar o modelo correspondente a cada subvisão.

Este mecanismo produz a construção automática de uma visualização constituida por diferentes objetos hierarquicamente compostos.

Exemplo de Instaciação:

A classe CollaborationView implementa uma visualização de grafo de fluxos de mensagens, solicitando ao seu modelo os nodos e elos a serem mostrados, quando recebe a mensagem *buildVisualPresentation*. Quando uma instância desta classe é criada, informa sua estratégia de distribuição através do método de classe *defaultLayoutStrategy*.

```
LuthierTopLevelView subclass: #CollaborationView
.....

defaultLayoutStrategy
        "Retorna a classe de estrategia de distribuiçao utilizada por esta visão"
        ^CollaborationLayoutStrategy

buildVisualPresentation
        "Cria a estrutura da visualização formada por objetos da classe ComponentView para os nodos e
        MethodCallView para os elos"

        layoutStrategy initialize. "Inicializa a estrategia"
        self clearAll. "Remove os componentes"

        "Cria as instâncias que comporão a visualização"

        model getNodes do: [:c | self add: (ComponentView new model: c) ].
```

```
model getLinks do: [:c | self addLink: (MethodCallView new model: c)].
self invalidate.
```

6.4.3 Agregando manipulação direta

Para cada visão que será sensível a eventos do mouse, ou teclado, é necessário agregar os manejadores de eventos adequados e os comandos associados, caso o evento ative alguma operação do modelo.

LuthierEventHandlerView fornece o comportamento necessário para associar instâncias de manejadores de eventos com uma visão dada, através dos métodos seguintes:

- ***installHandler: anEventHandler*** para agregar um manejador à lista de manejadores associados com a visão que recebe a mensagem.
- ***releaseHandler: anEventHandler*** para remover um dado manejador da lista de manejadores associados com a visão que recebe a mensagem.

Exemplo de Instanciação:

Por exemplo, a classe *CollaborationView* agrega, na sua inicialização, manejadores para o tratamento do deslocamento dos seus componentes (*DragHandler*) e um outro para a ativação de um comando de seleção de componentes, utilizando uma interface de banda elástica (*RubberBandHandler*).

```
initialize
        super initialize.
        self installHandler: (LuthierDragHandler new forButton:#left).
        self installHandler: ((LuthierRubberBandHandler new forButton: #left)
                command: LuthierSelectionCommand new).
```

Quando uma visão não deve continuar tratando um dado tipo de evento, então é necessário remover o manejador associado dessa visão.

```
visão releaseHandler: aHandler.
```

6.4.4 Criação de manejadores de eventos

A classe abstrata *LuthierEventHandler* define o comportamento genérico dos manejadores de eventos. Quando um manejador é criado, deve-se especificar o botão do mouse do qual trata eventos, através da mensagem:

```
handler forButton: { left|middle|right}
```

A implementação genérica provida pela classe abstrata para o tratamento de eventos verifica se o evento é do tipo e botão tratados pelo manejador. Subclasses desta classe só devem implementar o tratamento específico de cada tipo de manipulação de mouse, através da redefinição de todos, ou alguns, dos seguintes métodos:

- *eventStart: anEvent*, para realizar possíveis ações de inicialização, deve retornar *true* se o evento pode ser tratado.

- *eventDo: anEvent*, para realizar o tratamento continuo do evento. O método é invocado até que produz-se um novo evento de mouse.
- *eventEnd: anEvent*, para realizar ações de finalização ou executar o comando associado.

A hierarquia de subclasses atualmente provida por Luthier é a seguinte:

```
LuthierEventHandler ('view' 'command' 'running' 'handledEventPrototype')
        LuthierClickHandler ()
        LuthierDoubleClickHandler ()
        LuthierDragHandler ('startPoint' 'draggedView')
                LuthierRubberBandHandler ('stopPoint')
                LuthierShiftDragHandler ('stopPoint')
                        LuthierResizeHandler ()
        LuthierPressHandler ()
                LuthierMenuHandler ('options' 'commands' 'separators')
                        LuthierMenuDispatcher ('agent')
```

Um manejador pode interagir com a instância de *LuthierEvent* que recebe como argumento para requisitar a posição corrente do cursor, relativa a janela na qual está contida a visão. Esta solução evita que os manejadores dependam da existência de um controlador ou uma janela especifica.

Exemplo de Instanciação:

O exemplo apresenta um manejador para o deslocamento de figuras na tela. O método *eventStart* recupera qual a figura que se encontra no ponto onde o mouse foi pressionado e armazena esse ponto. O método *eventDo:* mostra um retângulo na tela utilizando uma primitiva que acompanha o movimento do cursor pelo tempo especificado. O método eventEnd: é chamado quando o botão do mouse e livrado, e faz com que a figura selecionada desloque-se até o ponto atual do cursor.

```
LuthierEventHandler subclass: #LuthierDragHandler
        instanceVariableNames: 'startPoint draggedView '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Luthier Event Handling'

eventStart: anEvent
        (draggedView := view viewAt: anEvent cursorPoint) isNil ifTrue:[ ^false].
        startPoint := anEvent cursorPoint.
        ^true

eventDo: anEvent
        Screen default
                displayShape: draggedView feedbackView
                at: anEvent sensor window globalOrigin + (view localPointToGlobal: anEvent
                        cursorPoint)
                forMilliseconds: 10

eventEnd: anEvent
        draggedView moveTo: anEvent cursorPoint
```

Um manejador de *click* só precisa redefinir o método *eventEnd*, já que o evento é considerado instantâneo. Neste caso, o manejador gera a execução do comando associado através da mensagem *execute*, implementada pela superclasse *LuthierEventHandler*.

```
LuthierEventHandler subclass: #LuthierClickHandler
```

```
eventEnd: anEvent
        self execute: anEvent cursorPoint.
        ^true
```

6.4.5 Implementação de comandos

Os comandos encapsulam a execução de operações implementadas por um modelo ou uma visão, permitindo separar o tratamento específico dos eventos de entrada, das operações específicas que devem ser executadas como conseqüência de cada manipulação.

A classe abstrata *Command* define os métodos abstratos *execute, execute:* e *execute:with:*, algum dos quais deve ser redefinido para cada comando específico a ser implementado por uma subclasse. A primeira variante serve aos fins de executar operações que não requerem da informação acerca da visão, a segunda variante recebe a visão sobre a qual foi realizada a manipulação, e a terceira um argumento adicional com as posições de início e fim da manipulação do mouse.

A implementação de um comando depende obviamente da função que deva implementar e das operações que devam ser disparadas no modelo.

Cada comando tem associado pelo menos um manejador de eventos e, possivelmente, um outro objeto, seu sujeito, o qual implementa as operações que dispara o comando.

Exemplos de Instanciação:

A classe *LuthierSelectionCommand* implementa um comando de seleção de componentes da visão que recebe com argumento contidos no retângulo que recebe como segundo argumento. O comando, recupera os modelos dos componentes selecionados e abre um novo *browser* sobre um abstrator de seleção que restringe o modelo aos componentes selecionados.

```
LuthierCommand subclass: #LuthierSelectionCommand
        instanceVariableNames: 'selection '
        .....
        execute: aView with: aRectangle
                | components messages |
                selection := (aView viewsInside: aRectangle) collect: [:v | v model].
                subject openOn: ((SelectionAbstractor on: aView model subject)
                                forComponents: components)
```

Esta classe é projetada para trabalhar em conjunto com um *LuthierRubberBandHandler* (associado com a visão que recebe como primeiro argumento) que gerencia a manipulação de seleção através de um retângulo variável sob controle do usuário:

```
aView installHandler: ((LuthierRubberBandHandler new forButton: #left)
        command: LuthierSelectionCommand on: CollaborationBrowser).
```

Um outro exemplo é um comando que executa uma operação de remoção seu sujeito, a qual elimina um objeto selecionado na visão. Uma vez executada a operação o comando instrui à visão a gerar-se novamente para refletir a mudança.

```
execute: aView
        subject remove: aView model.
        aView topView buildVisualPresentation.
```

6.4.6 Criando estratégias de distribuição de visões

A classe abstrata *LuthierLayoutStrategy* define o protocolo seguido por clientes que requerem de um algoritmo de distribuição espacial. Subclasses desta classe devem fornecer a implementação destes algoritmos através dos métodos:

- ***add*** para agregar a visão dada como argumento à estrutura da visão cliente, em uma posição determinada.
- ***addLink*** para agregar a visão de um elo dada como argumento à estrutura da visão cliente, em uma posição determinada.
- ***insert:model:*** para criar a instância da classe de visão que recebe como primeiro argumento e agregá-la à visão cliente, em uma posição determinada.
- ***insertLink:model*** para criar a instância da classe de elo que recebe como primeiro argumento e agregá-la à visão cliente, em uma posição determinada.
- ***relayoutGraphFor:*** para recalcular as posições correspondente de uma visão, tomando em consideração os componentes restantes.

Exemplos de Instanciação:

A classe *HorizontalLayoutStrategy* implementa um algoritmo de distribuição que organiza os elementos horizontalmente, dentro de uma caixa retangular, cujo limite horizontal é definido pela variável *xLimit*.
O algoritmo calcula a posição do componente a ser inserido em função da posição do último componente do cliente. Se a posição supera o limite estabelecido para o largo da apresentação do cliente, então cria uma nova linha, procurando o primeiro espaço disponível evitar a sobreposição de visões.

```
LuthierLayoutStrategy subclass: #HorizontalLayoutStrategy
instanceVariableNames: 'nextPosX nextPosY xLimit xSpace ySpace defaultPosY defaultPosX '
classVariableNames: ' '
poolDictionaries: ''
category: 'Luthier Layout Strategies'

add: aView
| b pos int fig |
pos := defaultPosX @ defaultPosY
clientObject components isEmpty
        ifFalse: [b := clientObject components last bounds.
                b corner x + xSpace > xLimit
                ifTrue: [ nextPosY := b corner y + ySpace.
                        nextPosX := defaultPosX]
                ifFalse: [nextPosX := b corner x + xSpace].
                int := clientObject components detect: [:c | c bounds intersects: (nextPosX @ nextPosY
                                                                extent: aView bounds extent)]
                                                        ifNone: [].
                [int notNil]
                        whileTrue:
                                [nextPosX := int bounds corner x + xSpace.
```

```
                int := clientObject components detect: [:c | c bounds intersects:
                        (nextPosX @ nextPosY extent: aView bounds extent)]
                                ifNone: [].
        int notNil ifTrue: [nextPosY := int bounds corner y + ySpace]]
        pos := nextPosX @ nextPosY.
        aView bounds: (0 @ 0 extent: aView bounds extent)].
fig:= clientObject add: aView at: pos.
^fig
```

Já o método encarregado de redistribuir os objetos do cliente quando tem se produzido alguma mudança, reorganiza a distribuição, a partir de uma visão dada, removendo os componentes do cliente, a partir dessa visão, e agregando-os novamente utilizando o algoritmo anterior.

```
relayoutGraphFor: aView

    | comps |
    comps := clientObject components copyFrom: (clientObject components indexOf: aView)
                                        to: clientObject components size.
    clientObject components removeAll:comps.
     comps do:[:c| clientObject add: c]
```

6.4.7 Redistribuição *lazy* de visões

Em muitas circunstâncias, uma visão composta é criada incrementalmente, agregando novos componentes na medida em que estes são criados durante a execução. Estas mudanças na estrutura requerem da redistribuição dos componentes, podendo ocorrer muitas vezes, antes da visualização final ser mostrada na tela. Para evitar a sobrecarga de computação imposta pela redistribuição de todos os elementos de uma visão cada vez que uma mudança se produz, *LuthierStrategyView* fornece um mecanismo que permite realizar a redistribuição de cada visão só uma vez, antes de ser visualizada.

Quando uma dada visão alterou sua área ocupada, ou seu conteúdo, informa para sua supervisão que precisa ser reorganizada, através da mensagem *relayoutNeeded:*. Esta mensagem é propagada para acima na hierarquia, até a visão de nivel superior. Esta visão mantém uma coleção com aqueles objetos que precisam ser reorganizados. Quando a visão de nivel superior é instruída para mostrar seu contúdo na tela, provoca a reorganização prêvia de todas aquelas visões que informaram a necessidade de reorganização, enviando-lhes a mensagem *relayout*.

```
relayoutNeeded: anObject
        (relayoutCollection includes: anObject)
                ifTrue: [relayoutCollection remove: anObject].
        relayoutCollection add: anObject


relayout
        [relayoutCollection isEmpty]
                whileFalse:[ relayoutCollection removeFirst relayout].
```

Exemplos de Instanciação:

Por exemplo, uma visão composta fornece um método *remove:* para remover um componente dado de sua lista de componentes. A remoção deste componente, provavelmente, alterará o tamanho que a visão ocupará, bem como a distribuição de outros componentes. Por esta razão, é necessário que a visão informe que precisa ser reorganizada através da mensagem ***relayoutNeeded***, enviando-se ela mesma como argumento.

```
remove: aModel
        | figure |
        figure := components detect:[:f| f model == aModel] ifNone:[^nil].
        components remove:figure.
        self relayoutNeeded: self.
        ^figure
```

antes da visão redesenhar-se receberá a mensagem *relayout*, a qual chamará a sua estratégia de distribuição para que reorganize os componentes. Desta forma, se 20 componentes foram removidos, a representação final será gerada com uma única ativação do algoritmos de redistribuição.

6.4.8 Especificação de contratos

6.4.8.1 Gerenciamento de dependências entre objetos

Este contrato especifica o relacionamento genérico entre dois objetos, um dos quais (o *dependente*) quer ser informado das mudanças de estado que acontecem no outro (o *modelo*). A propriedade "*attributes*" representa o armazenamento do estado interno do objeto *modelo*. Este contrato não diz nada a respeito do modo em que o *dependente* reage quando o *modelo* muda o seu estado, só indica que o *modelo* informará aos *dependentes* dessas mudanças. Como conseqüência, cada dependente receberá a mensagem *update*. Também fica estabelecido que o *modelo* fornecerá um método de acesso ao seu estado interno (*getState*).

O contrato também não estabelece a cardinalidade do relacionamento. O *modelo* pode ter zero ou mais *dependentes*, mas não é fixado de quantos *modelos* pode depender um objeto dado.

CONTRACT Model-Dependent

Model **supports** [

state: Value.

setState (value:Value) ⇒ Δ state; changed; {state = value}.

getState () ⇒ **returns** state.

changed () ⇒ ⟨|| d : d ∈ Dependents : d → update⟩.

addDependent (dependent: Dependent) ⇒ {dependent ∈ Dependent}.

removeDependent (dependent: Dependent) ⇒ { dependent ∉ Dependent }.

]

Dependents: **Set** (Dependent) **where each** Dependent **supports** [

update ⇒ model → getState.

addModel (model: Model) ⇒ model → addDependent (current).

]

END CONTRACT

6.4.8.2 Relacionamento genérico entre modelo e visões em MVC

Este contrato é uma especialização do contrato *Model-Dependent*, que descreve como estão relacionados os componentes da visualização (*View*) com os objetos aos quais representam (*Model*). Neste contrato, o *View* armazena uma referência ao seu *Model*, razão pela qual é implícito que cada *View* terá só um *Model* associado.

O invariante do contrato indica como deve reagir o *View* às mudanças do modelo, isto é, deve atualizar a sua apresentação, respeitando a propriedade que esta apresentação representa o estado do modelo.

CONTRACT Model-View

refines Model-Dependent (Model = Model, Dependent = View).

Model **supports** [

attribute: Value.

setAttribute (aValue:Value) ⇒ Δ attribute; changed; {attribute = aValue}.

getAttributeValue ⇒ **returns** attribute.

]

Views: **Set** (View) **where each** View **supports** [

model: Model.

update () ⇒ display.

display() ⇒ model → getValueAttribute; {View **presents** model.attribute}.

setModel (aModel: Model) ⇒ Δ model; **former**(model) → removeDependent (**current**);
buildVisualPresentation;
model → addDependent(**current**); {model = aModel}.

buildVisualPresentation()⇒ ϕ

]

invariant

Model.setAttribute (attrName, aValue) ⇒ ⟨ ∀ v : v ∈ Views : v **presents** Model.attribute⟩.

END CONTRACT

6.4.8.2.1 Relacionamento entre Componentes e Visões

Este contrato é uma especialização do relacionamento entre um modelo e sua visão, para levar em conta que o estado do *model* está representado em forma de atributos, aos quais é possível acessar através do seu nome.

CONTRACT Component-ComponentView

refines Model-View (Model = Component, View = ComponentView).

Component **supports** [

attributes: **Set** (<String,Value>)

setAttribute (attrName:String; aValue:Value) ⇒ Δ attributes; changed;
{<attrName,aValue> ∈ attributes ∧ !∃ otherValue, otherValue ≠ aValue, <attrName,otherValue> ∈ attributes}.

getAttributeValue (attrName: String) ⇒ **returns** aValue; {<attrName,aValue> ∈ attributes}.

]

Views: **Set** (ComponentView) **where each** ComponentView **supports** [

display() ⇒ model → getAttributeValue; { ComponentView **presents** model.attributes}.

].

invariant

Component.setAttribute (attrName, aValue) ⇒

{ ∀ v : v ∈ Views : v **presents** Component.attributes}.

Component.setAttribute (attrName, aValue) ⇒

{∀<attrName,aValue> : <attrName,aValue> ∈ Component.attributes :

!∃ otherValue, otherValue ≠ aValue ∧ <attrName,otherValue> ∈ Component.attributes}.

END CONTRACT

6.4.8.2.2 Relacionamento entre Contextos e Visões

Especialização do relacionamento entre um componente e sua visão, este contrato acrescenta as responsabilidades que um objeto composto por nodos e elos (o contexto) deve acrescentar para o acesso desses componentes, além dos atributos próprios, aos quais é possível acessar através do seu nome.

Este contrato é também um exemplo de uma situação que repete-se com freqüência: ainda que o contrato é entre dois participantes (*Contex* e *ContextView* neste casso), o contrato especifica a intervenção de outros tipos de componentes (o *Link* e o *Componente*, por exemplo). Isto é devido a que os participantes não estão projetados para se comunicar somente entre eles, mas sim para ser parte de um framework.

CONTRACT Context-ContextView

refines Component-ComponentView (Component = Context, ComponentView = ContextView)

Context **supports** [

contents: **Set** (Component).

links: **Set** (Link).

add (aComponent: Component) ⇒ Δ contents; changed; {aComponent ∈ contents}.

add-at (aComponent: Component; aPoint: Point) ⇒ Δ contents;
changed (aComponent, aPoint); {aComponent ∈ contents}.

add-before (aComponent: Component, anotherComponent: Component) ⇒ Δ contents;
{(aComponent ∈ contents) ∧ (anotherComponent ∈ contents) ∧ (aComponent ≠ anotherComponent)
∧ (!∃ thirdComponent : thirdComponent ∈ contents : component ≠ thirdComponent ≠
anotherComponent)}.

addLink (LinkComponent link) ⇒ Δ links; changed; {link ∈ links}.

getLinks () ⇒ **returns** links.

getNodes () ⇒ **returns** contents.

remove (aComponent: Component) ⇒ Δ contents; changed; {aComponent ∉ contents}]

ContextViews: **Set**(ContextView) **where each** ContextView **supports** [

display() ⇒ model → getAttributeValue(); model → getNodes(); model → getLinks()
{ContextView **presents** model.attributes ∧ model.contents ∧ model.links}.

]

invariant

Context.add (aComponent) ⇒ ⟨ ∀ v : v ∈ ContextViews : v **presents** Context.contents⟩.

Context.add-at (aComponent, aPoint) ⇒ ⟨ ∀ v : v ∈ ContextViews : v **presents** Context.contents⟩.

Context.add-before (aComponent, anotherComponent) ⇒ ⟨ ∀ v : v ∈ ContextViews : v **presents**
Context.contents⟩.

Context.addLink (link) ⇒ ⟨ ∀ v : v ∈ ContextViews : v **presents** Context.getLinks⟩.

Context.remove (aComponent) ⇒ ⟨ ∀ v : v ∈ ContextViews : v **presents** Context.contents ∧ Context.links⟩.

END CONTRACT

6.4.8.3 Organização hierárquica de visões

Este contrato especifica as responsabilidades dos participantes de qualquer visualização construida com o framework, a qual é organizada como uma hierarquia de instâncias que implementam diferentes visões da informação e o tratamento dos eventos de entrada.

CONTRACT Superview-Subview

Superview **supports** [

subviews: Subviews.

extent: Rectangle.

addSubview (subview) ⇒ Δ subviews { subview ∈ subviews;
subview.superview = **current** }.

remove (subview) ⇒ Δ subviews { subview ∉ subviews;
subview.superview ≠ **current** }.

topView () ⇒ **returns current**;

reconfig () ⇒ { ⟨ ∀s : s ∈ subviews : **includes** (extent, s.extent) ⟩ }.

relayoutNeeded (subview) ⇒ϕ

display () ⇒ { ⟨ ∀s : s ∈ subviews : **if** (**includes** (extent, s.extent)) **then** s → display ⟩ }.

containsPoint (aPoint: Point) ⇒ **returns** (**contains** (extent, aPoint) ∨
(∃ s : s ∈ subviews : contains (s.extent, aPoint))).

intersects (aRectangle: Rectangle) ⇒ **returns** (**intersects** (extent, aRectangle) ∨
(∃ s : s ∈ subviews : intersects (s.extent, aRectangle))).

handleEvent (event: Event) ⇒ view → handleEvent (event, cursorPoint);
{ view ∈ subviews ∧ contains (view.extent, cursorPoint) ∧
view → handleEvent (event, cursorPoint) ∧
(!∃ anotherView : anotherView ∈ subviews ∧ anotherView ≠ view :
(view ≀ anotherView) ∧ (contains (anotherView.extent, cursorPoint)) ∧
(anotherView → handleEvent (event, cursorPoint)) }.

absoluteOrigin (point: Point).

]

Subviews: **Set** (Subview) **where each** Subview **supports** [

container: Superview.

dx: Integer.

dy: Integer.

extent: Rectangle.

topView () ⇒ **returns** container → topView.

relayoutNeeded (subview) ⇒ container→ relayoutNeeded (subview).

display () ⇒ ϕ.

relayout () ⇒ ϕ.

handleEvent (event: Event; point: Point) ⇒ ϕ.

absoluteOrigin (point: Point) ⇒ **returns** container → absoluteOrigin (point).

]

END CONTRACT

O contrato seguinte refina o relacionamento definido pelo anterior, acrescentando o protocolo para gerenciar uma lista de visões que informaram que necesitam recalcular sua distribuição espacial. Esta redistribuição é realizada antes de produzir o desenho na tela.

CONTRACT TopLevelView-Subview

refines Superview-Subview (Superview = TopLevelView; Subview = Subview).

TopLevelView **supports** [

relayoutCollection: Subviews.

display⇒ relayout ;

⟨ ; s : s ∈ subviews : **if** (**includes** (extent, s.extent)) **then** s → display ⟩ .

relayoutNeeded (subview) ⇒ Δ relayoutCollection; { subview ∈ relayoutCollection }.

relayout () ⇒ Δ relayoutCollection; ⟨ ; s : s ∈ relayoutCollection: s → relayout ⟩;
{ relayoutCollection = ∅ }.

]

Subviews: **Set** (Subview) **where each** Subview **supports** [

]

END CONTRACT

6.4.8.4 Tratamento de eventos de mouse

O contrato estabelece as responsabilidades das visões e os manejadores quando um evento é gerado pelo usuário. As visões propagam para seus manejadores associados a mensagem *recognizeEvent*, sendo que o primeiro manejador que reconhece o evento será o encarregado de tratá-lo.

CONTRACT View-EventHandler

includes

Superview-Subview (Subview = View).

View **supports** [

eventHandlers: **Set** (EventHandler).

handleEvent (event: Event; point: Point) ⇒ **returns** result; { result =
(∃ handler : handler ∈ eventHandlers : handler→recognizeEvent (anEvent, current)) ∧ (!∃
anotherHandler : anotherHandler ∈ eventHandlers :
anotherHandler ≠ handler : anotherHandler → recognizeEvent (anEvent, current)) }.

installHandler (handler: EventHandler) ⇒ Δ eventHandlers; { handler ∈ eventHandlers }.

releaseHandler (handler: EventHandler) ⇒ Δ eventHandlers; { handler ∉ eventHandlers }.

]

EventHandler **supports** [

handledEvent: Event.

recognizeEvent (anEvent: Event; aView: View) ⇒ { anEvent = handledEvent }; Δ view;
returns handleEvent (anEvent), { view = aView }.

handleEvent (anEvent: Event) ⇒ eventStart (anEvent);eventDo (anEvent), eventEnd (anEvent).

requires subclass to support:

eventStart (anEvent)

eventDo (anEvent)

eventEnd (anEvent)⇒ {◊ (execute))

]

END CONTRACT

6.4.8.5 Execução de comandos

Um comando é executado por um manejador, o qual lhe envia alguma das três variantes da mensagem *execute*. A implementação destas mensagens depende de cada comando em particular, más no caso geral, ativarão alguma operação do seu sujeito.

CONTRACT Command-EventHandler

Command **supports [**

subject: Object.

handlers:Set(EventHandler).

eventHandler(eventHandler: EventHandler)⇒ **{** eventHandler ∈ handlers**}**.

execute() ⇒ **{** ◊ subject→ **perform(***operation***)}.**

execute (view: View) ⇒ **{** ◊ subject→ **perform(***operation***)}.**

execute (view: View; point: Point) ⇒ **{** ◊ subject→ **perform(***operation***)}.**

]

EventHandler **supports [**

view: View

button: Symbol.

command: Command.

forButonCommand(Symbol, Command) ⇒ Δ button. Δ command.

command→ eventHandler(**current**).

execute (point: Point) ⇒ **{**command ≠ Ø**}**;

if command → arguments =2 **then** command → execute (view, point);

else if command → arguments = 1 **then** command → execute (view);

else command → **execute**;

]

END CONTRACT

6.4.8.6 Tratamento de estratégias de distribuição espacial de visões

Uma visão com estratégia de distribuição delega para a estratégia a determinação das coordenadas que terá uma visão a ser inserida, bem como, solicita a redistribuição da visualização quando alguma mudança tem se produzido, no seu tamanho atribuído ou pelo deslocamento interativo de outras visões componentes.

CONTRACT LayoutView-LayoutStrategy

LayoutView **supports [**

links: **Set** (LinkView).

nodes: **Set** (ComponentView).

```
    insertLink (linkType: ViewType; model: LinkComponent) ⇒ Δ links;
        { (layoutStrategy → insertLink (linkType, model)) ∈ links }.

    insert (classFigure: ViewType; position: Point; model: Component) ⇒ Δ nodes;
        { ( layoutStrategy → insert ( classFigure, position, model )) ∈ nodes }.

    removeAll () ⇒ layoutStrategy → initialize.

    relayoutGraphFor (anObject: Object) ⇒ layoutStrategy → relayoutGraphFor (anObject).

    changeBounds (sender) ⇒ layoutStrategy → relayoutGraphFor (sender).

]

LayoutStrategy supports [

    clientObject: LayoutView.

    initialize () ⇒ ϕ.

    insertLink (type: ViewType; model: LinkComponent) ⇒
        clientObject → addSubview (newLink); returns newLink;
        { type (newLink) = ViewType; newLink.model = model }.

    insert (type: ViewType; position: Point; model: Component) ⇒
        clientObject → addSubview (newNode);
        returns newNode;
        {type (newNode) = ViewType; newNode.model = model;
        newNode.x ≠ nil; newNode.y ≠ nil }.

    relayoutGraphFor (anObject: Object) ⇒ { ⟨ ∀ n : n ∈ clientObject.nodes :
        Δ n.x ∧ Δ n.y ∧ n.x ≠ nil ∧ n.y ≠ nil ⟩ }.

]

includes

  Superview-Subviews ( Superview = LayoutView; Subview = ComponentView ).

invariant

  LayoutStrategy.insert (type, position, model) ⇒ ⟨ ∀ n : n ∈ clientObject.nodes :
    ( n.x ≠ nil ∧ n.y ≠ nil ) ⟩.

  LayoutStrategy.relayoutGraphFor (anObject) ⇒ ⟨ ∀ n : n ∈ clientObject.nodes :
    ( n.x ≠ nil ∧ n.y ≠ nil ) ⟩.

END CONTRACT
```

6.4.9 Definição de classes e conformação de contratos

Por razões de brevidade, a seguir descrevem-se as interfaces daquelas classes mais importantes, as quais apresentam diferenças com os contratos. As classes *LuthierCommand* e *Luthier EventHandler* respondem ao protocolo definido pelos respectivos contratos.

Class LuthierEventHandlerCompositeView

conforms to View **in** View-SubViews

conforms to SubView **in** View-SubViews

conforms to View **in** Model-View

LuthierEventHandlerView **supports:[**

attributes

subviews **as** contents
role: *Lista de componentes visuais*

extent **as** bounds
role: *mantém o tamanho da área ocupada pela visão.*

base methods

addSubview **(**subview**) as**
add: aView
role: *Agrega um componente aos seus componentes.*

remove: aView
role: *Remove um componente aos seus componentes.*

topView **()**
role: *Retorna o componente de topo da hierarquia de visões.*

model: aModel
role: *Fixa o modelo da visão e dispara o mecanismo de construção da estrutura da visão*

subclass might redefine:

display () **as** displayOn: aGraphicContext
role: *Produz a apresentação gráfica do receptor e propaga a mensagem entre seus componentes a mensagem para produzir a apresentação gráfica na tela.*

requires subclass to support:

buildVisualPresentation
role: *Construção da estrutura de sub-visões a partir da informação contida no modelo.*

]

conforms to View **in** View-EventHandlers

LuthierEventHandlerView **supports:[**

base methods

eventHandlers**: Set (**LuthierEventHandler**).**

handleEvent **(**event: Event; point: Point**) as**

handleEvent: anEvent in: aPoint
role: *Propaga o evento entre seus componentes até que algum event-handler o trate.*

installHandler **(**handler: EventHandler**)**
role: *Agrega um manejador de eventos ao receptor.*

releaseHandler **(**handler: EventHandler**)**
role: *Remove um manejador de eventos do receptor.*

]

conforms to View **in** Abstractor-View

LuthierEventHandlerView **supports:[**

base methods

regenerateVisualPresentation
role: *Dispara o mecanismo abstrato de construção da estrutura de sub-visões a partir da informação contida no modelo, após eliminar todos os componentes.*

]

End Class LuthierEventHandleCompositeView

Class LuthierStrategyView

inherits from LuthierEventHandlerView

conforms to LayoutView **in** LayoutView-LayoutStrategy

LuthierStrategyView **supports [**

attributes

links
role: *Mantém a lista de elos da visualização*

nodes
role: *Mantém a lista de nodos da visualização*

layoutStrategy
role: *Referência à estratégia de distribuição da visão*

base methods

insertLink (linkType: ViewType; model: LinkComponent)
as insertLink: linkType model: aLink
role: *Cria uma instancia da classe de elo dada pelo primeiro argumento e a agrega na estrutura da visão*

insertNode(linkType: ViewType; model: LinkComponent)
as insertNode: nodeType model: aNode
role: *Cria uma instância da classe de nodo dada pelo primeiro argumento e a agrega na estrutura da visão.*

addNode:aNode
role: *Agrega o argumento na estrutura da visão.*

addNode:aNode
role: *Agrega o argumento na estrutura da visão.*

clearAll
role: *Remove todas as sub-visões e inicializa a estratégia de distribuição*

relayoutGraphFor:anObject:
role: *Propaga a mensagem a sua estratégia de distribuição*

]

End Class LuthierStrategyView

6.5 *LuthierAbstractors*: O Sub-Framework de Abstrações

LuthierAbstractors implementa os mecanismos genéricos que habilitam a construção de abstrações, filtragem e seleção de informação independente das visualizações. O sub-framework também fornece o suporte para o gerenciamento de escalas de abstração externamente manipuláveis pelo usuário.

6.5.1 Estrutura geral

Os abstratores são projetados para agir como um intermediário entre as visões e os componentes do modelo de hipertexto. Do ponto de vista das visualizações, um abstrator tem o papel de modelo da visualização. Um abstrator substitui, transparentemente, um componente do hipertexto através da implementação da mesma interface e controlando o acesso aos seus atributos. Isto é, um abstrator comporta-se como um *proxy* [GAM 94] dos componentes da representação de hipertexto.

A FIGURA 6.9 apresenta o diagrama de objetos do sub-framework de abstrações. A classe abstrata *Abstractor* define o comportamento genérico dos abstratores. Cada abstrator terá associada uma instância de *AbstractionLevel*, a qual definirá o seu nível de abstração corrente; uma visão (genericamente representadas pela classe *LuthierView*) e um sujeito (representado genericamente pela classe *Subject*).

As visões solicitam informação a ser visualizada através do protocolo padrão *getNodes* e *getLinks*, definido pela classe *Component*. O abstrator retornará a informação correspondente de acordo com a sua funcionalidade e o nível de abstração corrente.

O mecanismo de escalas de abstração é definido pela classe *AbstractionLevel*, a qual provê o suporte para comparação de níveis de abstração simbólicos. A classe *ScaleView* implementa a interface com o usuário da escala, a qual pode ser manipulada pelo usuário para variar interativamente o grau de detalhe desejado na visualização corrente (a interface *default* provida é uma barra de deslizamento).

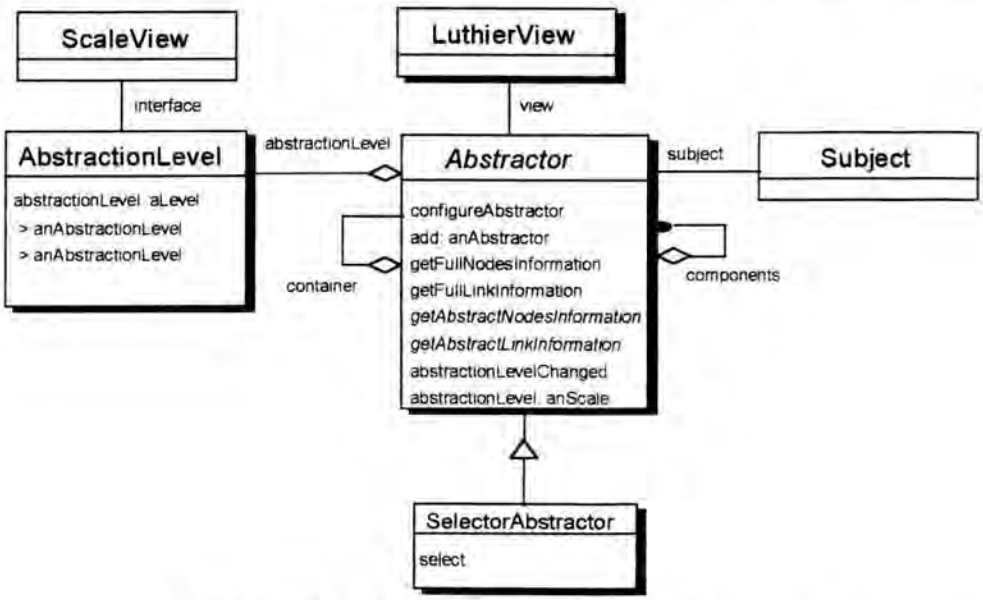


FIGURA 6.9- Modelo de Objetos de LuthierAbstractors

Quando o usuário produz uma mudança na escala de abstração global, a mensagem *abstractionLevelChanged* é propagada a todos os componentes para atualizar o nível de abstração corrente. Uma mudança no nível de abstração produzirá que o abstrator pai seja requerido para atualizar a visão global, a qual é informada através da mensagem *regenerateVisualPresentation*.

A FIGURA 6.10 apresenta um diagrama de interação abstrato, o qual representa a interação geral entre os abstratores, as visões, contextos e a mudança da escala de abstração.

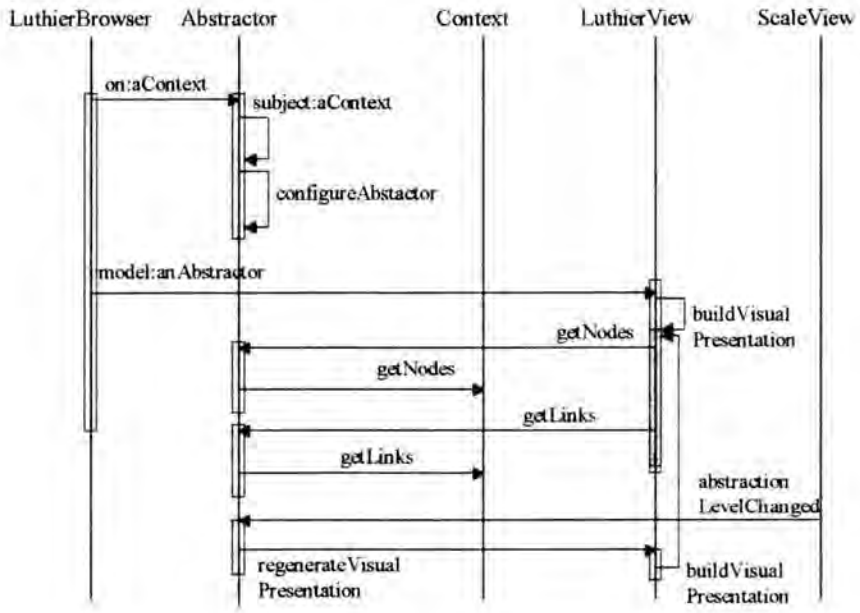


FIGURA 6.10- Coordenação entre abstratores, visões, contextos e escalas de abstração

6.5.2 Criação e composição de abstrações

Novos tipo de abstratores são definidos como subclasses da classe abstrata *Abstractor*. Estas subclasses, habitualmente, redefinem a implementação para os métodos hook *getFulltNodes Information*, *getFullLinkInformation*, *getAbstractNodesInformation* e *getAbstractLinkInformation*. Estes métodos são chamados pelos métodos *getNodes* e *getLinks*, os quais implementam a interface provida pelos contextos de *LuthierBooks*. A implementação destes métodos hook depende de cada tipo de abstrator.

Instâncias de abstratores são criadas através da mensagem *on: aSubject*, a qual cria a instância e fixa o sujeito do abstrator, disparando automaticamente o mecanismo de configuração descrito abaixo.

Ao implementar a mesma interface que os contextos, um abstrator pode receber como sujeito uma instância de um outro abstrator. Deste modo, abstratores que implementam diferentes funcionalidades, como por exemplo filtragem e seleção, podem ser compostos sobre um mesmo sujeito.

As mensagens enviadas pela visão associada serão propagadas entre os diferentes abstratores até alcançar, eventualmente, o contexto que contém a informação a ser visualizada.

Exemplos de Instanciação

Voltando ao exemplo da configuração dos *browsers*, pode observar-se a composição de dois abstratores, *FrameworkAbstractor* e *MessagePropertiesAbstractor*, com a instância de contexto a ser visualizada. O modelo da visão *topLelvelView* é a instância de *FrameworkAbstractor*, enquanto *topLevelView* é a visão associada com o abtrator .

```
configureBrowserFor: aContext
    "Cria a estrtura básica da visualização
    abstractor := FrameworkAbstractor on: ( MessagePropertiesSelector on: aContext ).
    topLevelView:= CollaborationView new.
```

```
abstractor view: topLevelView.
topLevelView model: abstractor.
```

6.5.3 Requisição de informação básica

As visões solicitam a seu modelo a informação a ser apresentada graficamente através de duas mensagens padrão: *getNodes* e *getLinks*. Alternativamente, também podem ser utilizadas as mensagens *allNodes* e *allLinks*. O comportamento o genérico destas mensagens implementa o mecanismo de controle do nível de abstração. Se o valor corrente da escala de abstração é maior que o nível de abstração representado pelo abstrator, então a informação completa é retornada, caso contrário, é retornada só a informação abstrata. O exemplo abaixo mostra a implementação destes métodos:

```
getNodes
        self abstractioLevel > self abstractionRepresented
                ifTrue:[ ^self getFullNodesInformation]
                ifFalse:[ ^self getAbstractNodesInformation].

getLinks
        self abstractioLevel > self abstractionRepresented
                ifTrue:[ ^self getFullLinkInformation]
                ifFalse:[ ^self getAbstractLinkInformation].
```

A implementação por falta dos métodos *getFullNodesInformation* retorna a lista de componentes do abstrator, enquanto a de *getAbstractNodesInformation* retorna uma lista vazia indicando que não há informação para ser visualizada no nível de abstração corrente.

Cada classe de abstrator deve fornecer a implementação do método de classe *abstractionRepresented*, o qual informa a escala de abstração na qual esse abstrator mostrará seu conteúdo.

Exemplos de Instanciação:

As implementações dos *hooks* de retorno de informação devem ser habitualmente redefinidas para implementar comportamentos específicos para cada tipo de abstrator. Por exemplo, um abstrator de subsistemas sempre deve retornar-se a si próprio, pois deve ser mostrado em qualquer nível de abstração:

```
Abstractor subclass: #SubsystemAbstractor
...
        abstractionRepresented
                ^#(subsystemAbstraction allAbstractions)

        getAbstractNodesInformation
                "Estando num nível de abstração maior, o abstrator retorna a si próprio"
                ^self
```

6.5.4 Criação de hierarquias de abstratores

O principal papel dos abstratores é fornecer um mecanismo uniforme para o controle do nível de detalhe de visualizações compostas, as quais representam o caso mais comum. Deste modo, a classe *Abstractor* está projetada para suportar uma organização hierárquica duplamente conectada de abstrator-subabstratores.

Para a criação de hierarquias de abstratores o método *configureAbstrator* é provido para ser redefinido por subclasses que precisam criar uma estrutura hierárquica correspondente com a estrutura de composição de seu sujeito. Quando um dado abstrator recebe seu sujeito, este método é automaticamente chamado pela implementação do método *setSubject*.

Exemplos de Instanciação:

Um abstrator de hierarquias redefine o método *configureAbstractor* para instalar abstratores sobre as classes que compõem a hierarquia. O método solicita ao sujeito, o qual é suposto ser uma hierarquia de classes, seus componentes criando, para cada um deles, um abstrator de classe. Cada abstrator de classe é agregado como componente do abstrator de hierarquia e recebe este como recipiente.

```
Abstractor subclass: #ClassHierarchyAbstractor
...
configureAbstractor
        | abstractor |
        subject getNodes
                do: [:c |
                        components add: (abstractor:= c ClassAbstractor on: c).
                        abstractor container: self]
```

6.5.5 Habilitação de abstratores e seleção de informação

Um abstrator pode estar, ou não, habilitado para passar informação para as visões. Quando um abstrator não está habilitado a implementação por falta do protocolo de transferência de dados retorna uma lista vazia, simulando o fato de que não há informação disponível para ser visualizada.

Esta funcionalidade é de grande utilidade para a visualização de informação resultante de um processo de seleção. Habilitando só os abstratores associados com a informação selecionada, a informação não selecionada não aparecerá na visualização.

Os métodos *enable* e *disable* são providos para a habilitação de abstratores. Por falta, a inabilitação é propagada para todos os componentes do abstrator sendo inabilitado, enquanto que a habilitação é propagada para todos os recipientes do abstrator sendo habilitado.

```
disable
        enabled := false.
        abstractionLevel disable.
        components do:[:c| c disable].
enable
        enabled := true.
        abstractionLevel enable
        container notNil ifTrue:[ container enable].
```

Um tipo de abstração não estrutural é representada pelos seletores. Os seletores permitem especificar um critério de seleção da informação a ser visualizada, como por exemplo, visualizar só aquelas classes que estão relacionadas através de mensagens que ativam métodos definidos como abstratos, em alguma superclasse. Estes critérios de seleção podem também ser manipulados interativamente pelo usuário, fornecendo desta forma a possibilidade que uma mesma apresentação seja variada no seu nível de detalhe de acordo com o tipo de informação desejada pelo usuário.

A classe *SelectorAbstractor* define o mecanismo genérico que habilita a seleção de um conjunto de objetos de acordo a um critério de seleção estabelecido pela subclasse de seletor utilizada. Quando uma visão solicita a informação do modelo, o seletor devolverá aquela informação que foi

selecionada. A implementação atual desta classe suporta a seleção de nodos e relacionamentos entre nodos de um grafo.

Exemplos de Instanciação:

- A classe *MessagePropertiesSelector* implementa a seleção de elos de acordo com um bloco especificado como argumento. Por exemplo, o bloco *[:link| link origin methodType = #abstract]* especifica a seleção de aqueles elos representando mensagens cuja origem é um método abstrato. Esta classe fornece a implementação mostrada abaixo do método que retorna os elos, o qual chama um método privado encarregado de aplicar o bloco a cada elo retornado pelo sujeito.

```
getFullLinkInformation
        ^self selectVisibleMessages: subject getLinks.

selectVisibleMessages: aCollection
        "Seleciona aqueles links que satisfazem o criterio de seleção de propriedades
        expressado pelo bloco"

        | sb |
        aCollection isNil ifTrue:[^#()].
        sb := selector selectBlock.
        ^aCollection select: [:k | k satisfies: sb].
```

- Uma classe *MessageKnowledegeAbstractor* é programada para habilitar só a visualização das mensagens e métodos informados pelo seu sujeito. Para isto redefine o método abstrato *getFullLinkInformation* para retornar as mensagens, habilitando os abstratores associados com a origem e o destino. O mecanismo de propagação ascendente na hierarquia da habilitação faz com que todos os abstratores que contém cada método seja habilitados recursivamente. Desta forma, a visualização resultante mostrará, por exemplo, as classes, os métodos associados pelas mensagens recebidas. Se o sujeito deste abstrator é uma instância da classe *MessagePropertiesSelector*, então a visualização poderá ser controlada pelo usuário através da especificação do bloco de seleção, reconfigurando-se automaticamente para mostrar só a informação recebida.

```
getFullLinkInformation
        | messages |
        self topComponent disable. "Inabilita todos os abstratores da hierarquia"
        messages := subject getLinks.
        messages
                do:
                        [:m |
                        self abstractorOf:( m origin) enable.
                        self abstractorOf:( m target) enable].
        ^messages
```

6.5.6 *Zoom* Semântico: Definição de escalas de abstração

Uma escala de abstração define uma seqüência ordenada de conceitos que representam diferentes níveis de abstração de uma hierarquia. Por exemplo, a escala abaixo é utilizada para definir o grau de detalhe da visualização de subsistemas:

(subsystemAbstracion abstractHierarchy abstractMethod concreteMethod concreteHierarchy)

A seleção de algum nivel nesta escala definirá que tipo de informação a visualização receberá para ser apresentada. Isto é, se o nível de abstração selecionado é *abstractHierarchy* a visualização só receberá a informação de quais os subsistemas e as classes do topo das hierarquias que

compõem cada subsistema. Após, se o nível de abstração selecionado é *abstractMethod*, a visualização receberá os subsistemas, as classes abstratas e os métodos abstratos definidos nessas classes.

A escala abaixo, por sua vez, estabelece uma ordem onde os métodos são mostrados só no último nível de detalhe da visualização:

(subsystemAbstracion abstractHierarchy concreteHierarchy abstractMethod concreteMethod)

O mecanismo de comparação de níveis de abstração simbólicos de escalas de abstração é implementado pela classe *AbstractionLevel*. Uma instância de *AbstractionLevel* mantém o valor corrente do nível de abstração do seu abstrator associado. Cada abstrator tem a sua própria instância desta classe, de tal modo, que é possível variar o nível de abstração de cada abstrator de forma independente. Desta forma é possível implementar, por exemplo, mecanismos do *zoom* semântico baseados no foco de atenção do usuário.

No caso de abstratores compostos hierarquicamente, o nível de abstração de uma visualização é comum, e as mudanças no nível superior são propagadas para todos os componentes.

Um abstrator é informado da escala de abstração que deverá respeitar através da mensagem *abstractionScale*: <escala>, o qual é propagado para todos os abstratores componentes da hierarquia.

A classe *ScaleView* implementa a interface com o usuário da escala, a qual pode ser manipulada pelo usuário para variar interativamente a grau de detalhe desejado na visualização corrente.

Exemplo de Instanciação:

Este exemplo completa o exemplo de configuração do browser apresentado acima, mostrando como o abstrator é configurado externamente com a escala de abstração que manejará.

```
configureBrowserFor: aContext
        "Cria a estrtura básica da visualização
        abstractor := FrameworkAbstractor on: ( MessagePropertiesSelector on: aContext ).

        abstractor abstractionScale:#(subsystemAbstracion abstractHierarchy
                                        concreteHierarchy abstractMethod concreteMethod)

        topLevelView:= CollaborationView new.
        abstractor view: topLevelView.
        topLevelView model: abstractor.
```

6.5.7 Geração e configuração dinâmica de abstrações

Existem casos nos quais o tipo de abstrator utilizado depende do nível de abstração corrente. Por exemplo, a visualização de um framework em termos de um grafo, pode estar formada por diferentes níveis de detalhe, como relacionamentos entre classes a nível de classes abstratas ou a nível classe concretas, até o nível de mensagens trocadas entre métodos. Isto implica na necessidade de selecionar, ou gerar, diferentes tipos de relacionamentos a serem visualizados, bem como adequar a visualização correspondente dos nodos e dos relacionamentos. Este problema pode ser resolvido programando *ad-hoc* o tipo de informação que o abstrator deve retornar em cada caso. Entretanto, uma solução mais elegante e reutilizável é criar abstratores específicos para cada nível de abstração envolvido e delegar para eles a seleção da informação a ser visualizada. Cada tipo de abstrator pode ser instanciado dinamicamente de acordo com as mudanças na escala de abstração.

O mecanismo de configuração pode ser utilizado para tal fim.

Exemplo de Instanciação:

A classe *FrameworAbstractor* delega a seleção de elos a classes *AbstractKnowledgeAbstractor*, *ConcreteKnowledgeAbstractor* e *MessageKnowledegeAbstractor*, as quais são instanciadas pelo método privado *setKnowledgeAbstractor*, quando se produz uma mudança na escala de abstração.[2]

```
HierarchyAbstractor subclass: #FrameworkAbstractor
        instanceVariableNames: 'knowledgeAbstractor '
.....

setKnowledgeAbstractor
        "Seleciona o tipo de abstrator adequado ao nível corrente"
        | knowledgeAbstractorClass |

        self abstractionLevel = # abstractHierarchy
                ifTrue:[ knowledgeAbstractorClass := AbstractKnowledgeAbstractor]
                .....
        knowledgeAbstractor := knowledgeAbstractorClass on: self subject.
        knowledgeAbstractor container:self.

getFullLinkInformation
                "Delega a solicitação de links ao abstrator corrente"
                ^knowledgeAbstractor getLinks
```

A classe *AbstractKnowledgeAbstractor*, por exemplo, implementa o algoritmo que deduz os relacionamentos abstratos entre classes a partir da informação das mensagens entre classes. Estes relacionamentos sintetizam todas as mensagens trocadas entre hierarquias de classes, desde a raiz, incluindo todas as subclasses.

6.5.8 Especificação de contratos

6.5.8.1 Gerenciamento de hierarquia de abstratores

Este contrato especifica o protocolo de propagação de mensagens que estabelece-se entre um conjunto de abstratores organizados em uma hierarquia de abstrator-subabstratores. O invariante do contrato estabelece que um abstrator pode estar contido por um abstrator que o tem com componente.

CONTRACT Abstractor-SubAbstractor

Abstractor **supports [**

components: **Set** (SubAbstractor).

add (anAbstractor) → Δ components;

{ anAbstractor ∈ components ∧ anAbstractor.container = **current)**.

addFirst (anAbstractor) → Δ components;

{ anAbstractor ∈ components ∧

(∀ a : a ∈ components : a ≠ anAbstractor ⇒ anAbstractor ⟨ a);
anAbstractor.container = **current}**.

---

[2] A implementação real deste exemplo é muito mais complexa e generalizada, mas foi simplificada para ilustrar o exemplo.

```
    setSubject (object: Object) → configureAbstractor;
                                { type (object) = SubAbstractor ⇒ object.container = current }.
    remove (anAbstractor) → Δ components;
                                { anAbstractor ∉ components ∧ anAbstractor.container ≠ current.}.
    configureAbstractor () → ϕ "Hook para subclasses que necessitam instalar abstratores sobre componentes do sujeito"
                                { ◊ Δ components}
    abstractorOf (aSubject: Model) → returns abstractor;
          { (abstractor = ∅) ∨ (∃ subabstractor : subastractor ∈ components :
          abstractor = subabstractor → abstractorOf (aSubject))}.
    enable () → <|| c : c ∈ components : c → enable >.
    disable () → <|| c : c ∈ components : c → disable >.
    abstractionLevel (aLevel) ⇒ <|| c : c ∈ components : c → abstractionLevel (aLevel)>.
    decreaseAbstractionLevel () ⇒ <|| c : c ∈ components : c → decreaseAbstractionLevel >.
    increaseAbstractionLevel () ⇒ <|| c : c ∈ components : c → increaseAbstractionLevel >.
    topComponent () ⇒ returns current.
    release () → <|| c : c ∈ components : c → release >.
]
SubAbstractor supports [
    subject: Model.
    container: Abstractor.
    view: View.
    abstractorOf (aSubject: Model) → returns abstractor;
          {abstractor = ∅ ∨ (aSubject = subject ∧ abstractor = current )}.
    enable () → Δ enabled; {enabled = true}.
    disable () → Δ enabled; {enabled = false}.
    abstractionLevel (aLevel) ⇒ abstractionLevelChanged.
    decreaseAbstractionLevel (aLevel) ⇒ abstractionLevelChanged.
    increaseAbstractionLevel(aLevel) ⇒ abstractionLevelChanged.
    topComponent () ⇒ returns container → topComponent.
    regenerateVisualPresentation () ⇒ if (view = ∅) then
          container → regenerateVisualPresentation.
    release () → ϕ      { ◊ components = ∅ }.
]
invariant
  SubAbstractor.container = Abstractor ⇔ SubAbstractor ∈ Abstractor.components .

END CONTRACT
```

6.5.8.2 Gerenciamento de escalas de abstração

Este contrato especifica o mecanismo de consistência entre a interface da escala de abstração e os abstratores. Quando o usuário produz uma mudança na escala de abstração global, a mensagem *abstractionLevelChanged* é propagada a todos os componentes para atualizar o nível de abstração corrente. Uma mudança no nível de abstração produzirá que o abstrator pai seja requerido para atualizar a visão global, a qual é informada através da mensagem *regenerateVisualPresentation*.

```
CONTRACT Abstractor-AbstractionLevel-ScaleInterface

  Abstractor supports [
    enabled: Boolean.
    level: AbstractionLevel.
    abstractionControl:ScaleInterface
    view: LuthierView.
    enable () ⇒ Δ enabled; level → enable; {enabled = true}.
    disable () ⇒ Δ enabled; level → disable; {enabled = false}.
    increaseAbstractionLevel () ⇒ level → incrementLevelBy (-1);
                                        abstractionLevelChanged.
    decreaseAbstractionLevel () ⇒ level → incrementLevelBy (1);
                                        abstractionLevelChanged.
    abstractionLevel (aLevel: Integer) ⇒ level → changeLevelTo (aLevel)
    interactiveAbstractionLevelChange ⇒ abstractionLevel (abstractionControl.value).
                                        abstractionLevelChanged.
    abstractionLevelChanged ⇒ view → regenerateVisualPresentation.
  ]
  AbstractionLevel supports [
    enabled: Boolean.
    abstractionScale: Set (Symbol).
    currentLevel: Integer
    enable () ⇒ Δ enabled; { enabled = true }.
    disable () ⇒ Δ enabled; { enabled = false }.
    changeLevelTo (aLevel: integer) ⇒ Δ currentLevel;
          {currentLevel = max (aLevel, size (abstractionScale))}.
```

```
    incrementLevelBy (aLevel: integer) ⇒ Δ currentLevel;
            { (currentLevel = former (currentLevel) + aLevel ∧
            1 ≤ currentLevel ≤ size (abstractionScale)) ∨
            (currentLevel = 1 ∧ former (currentLevel) + aLevel < 1) ∨
            (currentLevel = size (abstractionScale) ∧
            former (currentLevel) + aLevel > size (abstractionScale)) }.

  ScaleInterface supports [

    changed(newLevel: Integer) ⇒ Abstractor. interactiveAbstractionLevelChange.

  ]

  invariant

    AbstractionLevel.changeLevelTo (anIncrement) ⇒
      {1 ≤ AbstractionLevel.currentLevel ≤ size (AbstractionLevel.abstractionScale) }.

    AbstractionLevel.incrementLevelBy (anIncrement) ⇒
      { 1 ≤ AbstractionLevel.currentLevel ≤ size (AbstractionLevel.abstractionScale) }.

    Abstractor.enable () ⇒ { Abstractor.enabled = AbstractionLevel.enabled }.

    Abstractor.disable () ⇒ { Abstractor.enabled = AbstractionLevel.enabled }.

  includes

    Abstractor-SubAbstractor ( Abstractor = Abstractor, SubAbstractor = Abstractor).

END CONTRACT
```

6.5.8.3 Coordenação entre sujeitos e abstratores

Este contrato especifica o protocolo da colaboração entre sujeitos e abstratores, estendendo o contrato genérico entre modelos e dependentes, além de incluir o protocolo do contrato que define o relacionamento entre componentes de hipertexto. Os abstratores se alistam como dependentes do seu sujeito para serem informados das mudanças nele produzidas. As mensagens que um abstrator não trata diretamente são propagadas para seu sujeito através da mensagem *doesNotUnderstand:*, implementando deste modo, a substituição transparente do modelo real das visões. Adicionalmente, um abstrator fornece métodos de utilidade para procurar o abstrator associado com um dado objeto, bem como para a comparação de abstratores.

```
CONTRACT Subject-Abstractor

  refines Model-Dependent (Model = Subject; Dependent = Abstractor)

  Subject supports []

  Abstractor supports [

    subject: Model.

    doesNotUnderstand (aMessage: Message) ⇒ subject → perform (aMessage).

        As mensagens que o Abstractor não entende, são propagadas ao Model

    setSubject (object: Model) ⇒ Δ subject; subject → addDependent (current);
        configureAbstractor; {subject = object}.

    abstractorOf (aSubject: Model) ⇒ returns abstractor;
        {abstractor = ∅ ∨ (aSubject = subject ∧ abstractor = current )}.
```

```
    isKindOf (aClass: Class) ⇒ returns result;
          { result = (type (current) = aClass) ∨ (type (subject) = aClass) }.
    equal (object: Object) ⇒ returns result; { result = (object = current) ∨ (object = subject) }.
    release () ⇒ subject → removeDependent (current); {subject = ∅}.
]
includes Context-Component-Link (Component = Subject)
END CONTRACT
```

6.5.8.4 Coordenação entre visões e abstratores

Um abstrator instrui a sua visão para regenerar-se, quando tem-se produzido alguma mudança, através da mensagem *regenerateVisualPresentation*. Esta mensagem dispara o mecanismo abstrato de construção da representação visual, *buildVisualPresentation*. Este, eventualmente, solicita a informação a ser apresentada graficamente através de duas mensagens padrão: *getNodes* e *getLinks*. O comportamento genérico destas mensagens implementa o mecanismo de controle do nível de abstração. Se o valor corrente da escala de abstração é maior que o nível de abstração representado pelo abstrator, então a informação completa é retornada, caso contrário é retornada só a informação abstrata.

```
CONTRACT Abstractor-View
  refines Context-ContextView (Component = Abstractor; ComponentView = View);
  Abstractor supports [
    view: View
    subject: Component
    abstractionLevel: AbstractionLevel.
    regenerateVisualPresentation () ⇒ view → regenerateVisualPresentation.
    getNodes() ⇒ if(abstractioLevel > abstractionRepresented)
                      then return getFullNodesInformation;
                      else return getAbstractNodesInformation.
    getLinks()⇒ if(abstractioLevel > abstractionRepresented)
                      then return getFullLinkInformation;
                      else return getAbstractLinkInformation.
    getAbstractNodesInformation ⇒ϕ {◊( return ∅)}
    getAbstractLinkInformation ⇒ ϕ {◊( return ∅)}
    getFullNodesInformation() ⇒ return ( Set(Abstractor)).
    getFullLinkInformation()⇒ return ( Set(Abstractor)).
  ]
  View supports [
    model(aModel: Model) ⇒ Δ model;
          model → addDependent(current); buildVisualPresentation;
          {model = aModel; model.view = current}.
```

regenerateVisualPresentation () ⇒buildVisualPresentation

buildVisualPresentation()⇒ { ◊ (model → getNodes() ∧ model → getLinks())}

]

END CONTRACT

6.5.9 Definição de classes e conformação de contratos

Class Abstractor

conforms to Abstractor **in** Abstractor-SubAbstractor

conforms to Sub-Abstractor **in** Abstractor-SubAbstractor

Abstractor **supports**[

atributes

components: Set(Abstractor)
role: *Referência à coleção de abstratores componentes.*

container: Abstractor
role: *Referência ao abstrator recipiente*

subject: Object
role: *Referência ao objeto da representação que o abstrator representa.*

view: LuthierView
role: *Referência à visão associada*

enabled:Boolean
role: *Define se o abstrator está habilitado ou não.*

template methods

getNodes
role: *Retorna a informação do sujeito de acordo com o nível de abstração atual.*

getLinks
role: *Retorna os elos do sujeito de acordo com o nível de abstração atual.*

base methods

on:(Subject)
role: *Cria um abstrator sobre o sujeito especificado como argumento*

setSubject (Component)
role: *Seta o sujeito do abstrator*

abstractorOf (Subject)
role: *Retorna o abstrator na hierarquia de componentes correspondente com o sujeito dado. A mensagem propaga-se recursivamente na hierarquia de abstratores até que o abstrator é encontrado*

topComponent
role: *Retorna o abstrator topo da hierarquia*

enable
role: *Habilita um abstrator a retornar seu conteúdo*

disable
role: *Inabilita um abstrator de retornar seu conteúdo.*

abstractionLevel (AbstractionLevel)
role: *Seta o nível de abstração associado ao abstrator*

decreaseAbstractionLevel
role: *Decrementa o nível de abstração associado*

increaseAbstractionLevel
role: *Incrementa o nível de abstração associado*

isKindOf (Class)
role: *Testa se o abstrator ou seu sujeito herdam de uma dada classe*

equal (object: Object)
role: *Testa se o abstrator ou seu sujeito são iguais a um dado objeto*

release
role: *Elimina o recursivamente a referência aos sujeitos para sua liberação pelo garbage collector*

regenerateVisualPresentation
role: *Instrui à visão associada para regenerar a apresentação visual quando tem se produzido alguma mudança.*

doesNotUnderstand **(**aMessage: Message**)**
role: *As mensagens que o Abstractor não entende, são propagadas ao Modelo.*

subclasses might redefine:

add:(Abstractor)
role: *Agrega um abstrator aos componentes*

addFirst:(Abstractor)
role: *Agrega um abstrator aos componentes com primeiro elemento*

remove (anAbstractor)
role: *Remove um abstrator da lista de componentes*

configureAbstractor
role: *Hook para subclasses que necessitam instalar abstratores sobre componentes do sujeito*

subclass might redefine:

getAbstractNodesInformation
role: *Deve retornar os componentes visíveis de acordo com o nivel de abstração atual.*

getAbstractLinksInformation
role: *Deve retornar os elos visíveis de acordo com o nível de abstração atual.*

getFulNodesIInformation
role: *Retorna a lista de componentes*

getFulLinksIInformation
role: *Retorna a lista de elos*

]
End Class Abstractor

Class AbstractionLevel

conforms to AbstractionLevel **in** Abstractor-AbstractionLevel-ScaleInterface

AbstractionLevel **supports:[**

attributes

enabled: Boolean.

abstractionScale: **Set** (Symbol).

currentLevel: Integer.

base methods

enable
role: *Habilita a comparação entre níveis de abstração.*

disable
role: *Inibe a comparação entre níveis de abstração, retornando estas falso.*

changeLevelTo **(Integer)**
role: *Muda o nível corrente de abstração.*

incrementLevelBy **(Integer)**
role: *Incrementa o nível corrente de abstração.*

>(AbstractionLevel)

<(AbstractionLevel)

>=(AbstractionLevel)

<=(AbstractionLevel)

=(AbstractionLevel)
role: *Comparam o nível corrente de abstração com a escala de abstração recebida como argumento*

]
End Class AbstractionLevel

6.6 Utilização de Luthier

Nas seções anteriores descreveram-se, sinteticamente, as caraterísticas principais dos quatro sub-frameworks que compõem Luthier. O objetivo dessa descrição é fornecer uma referência do projeto detalhado e das principais decisões de projeto, bem como mostrar os mecanismos internos que habilitam a construção de especializações muito específicas. Nesta seção descrevem-se algumas características da biblioteca de componentes implementados com Luthier.

6.6.1 Visões, abstratores é meta-objetos

No caso geral, para construção de ferramentas, a biblioteca de componentes prontos desenvolvida, tem demonstrado a necessidade de programar pouco comportamento novo. Basicamente, para a construção de diferentes visualizações que utilizam a informação coletada por meta-objetos projetados para tal fim, é necessário só definir uma subclasse de *LuthierBrowser* para especificar a composição de abstratores e a interface da ferramenta.

No caso das visões, em geral é necessário programar novas classes que implementem desenhos de figuras diferentes aos existentes na biblioteca. Esta é uma tarefa que pode resultar complexa dependendo da complexidade da visualização, devido às características do MVC, particularmente na atribuição de espaço das visões. No caso das classes de manipulação direta não necessário no caso geral programar novas classes, exceto para o caso de comandos que implementem operações específicas. Neste caso, a programação é geralmente trivial.

O conjunto de abstratores definidos na biblioteca (FIGURA 6.11) tem mostrado ser suficiente para implementar um conjunto bastante amplo de visualizações baseadas em grafos. Entretanto, a

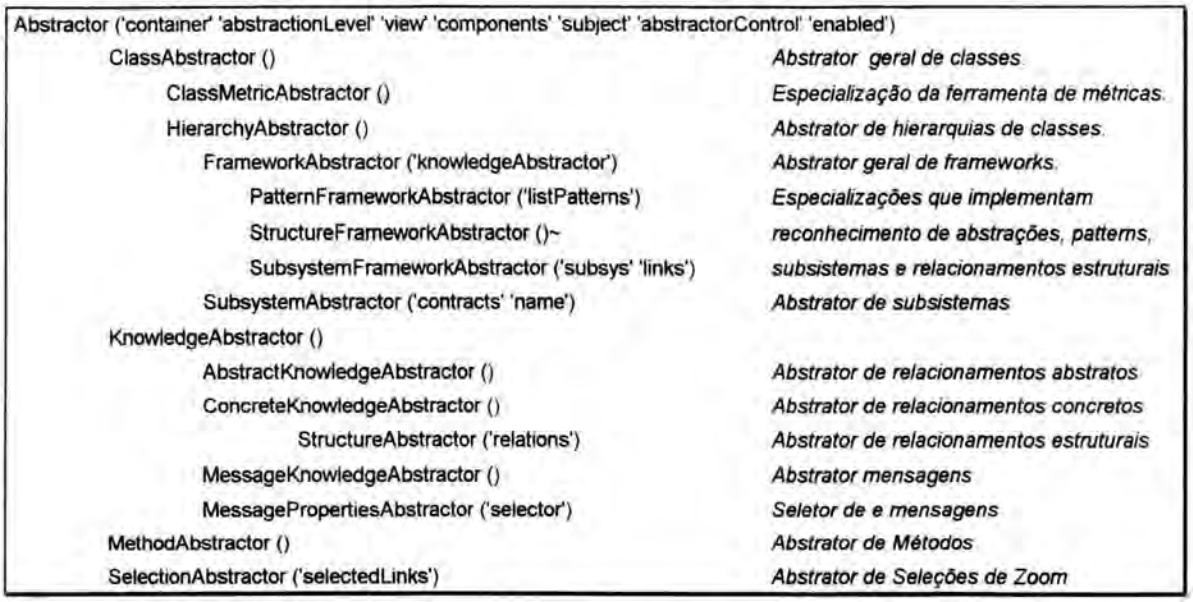


FIGURA 6.11- Hierarquia de abstratores concretos existentes na biblioteca de Luthier

programação de novos abstratores é uma tarefa relativamente simples, pois é necessário redefinir só um conjunto reduzido de métodos.

A nível de meta-objetos, a funcionalidade de cada classe de meta-objetos depende, obviamente, de cada tipo de aplicação. Deste modo, é difícil fornecer uma biblioteca muito rica de comportamentos reutilizáveis. A FIGURA 6.12 apresenta a hierarquia de classes de meta-objetos existente, até hoje, na biblioteca de Luthier. Muitas destas classes foram programadas para implementar funcionalidade específica de análise de programas, outras, como por exemplo *SoundMetaObject*, implementam funcionalidade que pode ser acrescentada a qualquer programa.

6.6.2 Manipulação de texto

Luthier fornece um suporte de texto para a construção de livros, o qual não requer a programação de novas subclasses. Este suporte oculta as características próprias da implementação de texto de Smalltalk, as quais são complexas de utilizar para um usuário não especialista.

```
LuthierEventHandlerPart ('dependents' 'dx' 'dy' 'fixed' 'hideFlag' 'eventHandlers')
        LuthierText ('preferredBounds' 'text' 'emphasis' 'textAttributes')
                LuthierLabel ()
                LuthierPluggableText ('getTextMsg')
```

A classe *LuthierText* fornece o comportamento necessário para a manipulação de texto com estilos associados, os quais podem ser variados dinamicamente. A classe *LuthierPluggableText* implementa o comportamento que acrescenta um nível de indireção para a obtenção do texto a ser visualizado. Esta classe é parametrizada com uma mensagem, através da qual, é solicitado o texto ao objeto definido como modelo. Isto permite a geração de texto formatado a partir de qualquer objeto que responda com uma cadeia de caracteres a uma mensagem dada, como por exemplo, o código de um método.

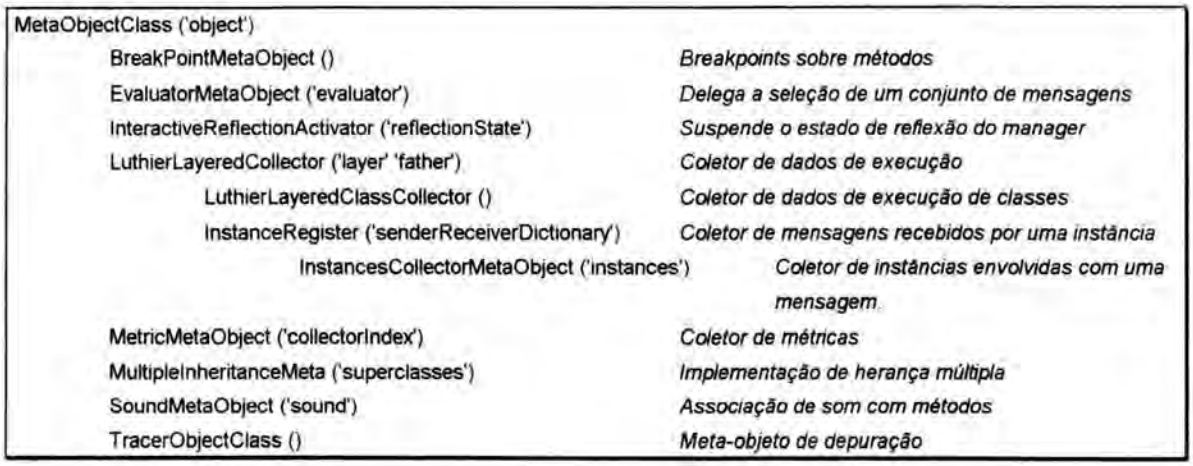


FIGURA 6.12 - Hierarquia de meta-objetos da biblioteca de Luthier

6.6.2.1 Estilos de texto

A classe *LuthierStyle* implementa o comportamento que permite a definição dinâmica de estilos de formatado de texto. Os formatos implementados são os formatos habituais existentes nos processadores de texto, isto é, justificação, fonte, ênfase de caracteres, espaçado, etc. Diferentes estilos podem ser criados associando-lhes um nome, como mostram os exemplos seguintes:

```
Cria um estilo chamado Normal, justificado e com uma indentação na primeira linha de
20 pontos.
(LuthierStyle newStyle: 'Normal') justified;
        firstIndent: 20;
        restIndent: 0.

Cria um estilo chamado Chapter Title, centrado, enfatizado, fonte grande e serifa, com
espaço de 12 pontos antes e depois.
(LuthierStyle newStyle: 'Chapter Title') centered;
        characterAttributes: #(#bold #large #serif);
        spaceBefore: 12;
        spaceAfter: 12.
```

A associação de um dado estilo com uma instância de *LuthierText* é realizada enviando a esta instância a mensagem *style:*. O exemplo abaixo mostra a criação de um novo texto *Capítulo 1* ao qual é associado um estilo um estilo chamado *Chapter Title*.

```
(LuthierText withText: 'Capítulo 1')
        style: (LuthierStyle styleNamed: 'Chapter Title')
```

6.6.3 Visões *Template*

Os *templates* de visões são um mecanismo importante para evitar a programação de classes de visões específicas. Um *template* representa um visão normal, a qual utiliza um bloco para construir sua estrutura interna. Esta estrutura pode estar recursivamente composta por outros *templates*, o qual permite definir combinações de visões simples para formar visões mais complexas.

Os templates são de grande utilidade para permitir variar interativamente o tipo de visão que será utilizada para um dado modelo, evitando codificar nomes de classes específicas em visões gerais, como por exemplo, uma visão de página de um livro.

O exemplo abaixo mostra a definição de um template denominado *MethodDescription*, o qual cria uma visão composta de três textos que representam a definição de um método Smalltalk (cabeçalho, comentário, corpo). Através deste template evita-se a criação de uma subclasse especifica para implementar a visão composta, bem como, pode ser utilizado por um outro template para inserir a definição textual de um método em uma visualização qualquer.

```
LuthierTemplate newTemplate: 'MethodDescription'
        template: [:container :model |
                container addNode: ((LuthierPluggableText new)
                                                textAccessor: #definition;
                                                model: model;
                                                style: (LuthierStyle styleNamed: 'Definicion')).
                container addNode: ((LuthierPluggableText new)
                                                textAccessor: #comment;
                                                model: model;
                                                style: (LuthierStyle styleNamed: 'Comentario')).
                container addNode: ((LuthierPluggableText new)
                                                textAccessor: #body;
                                                model: model;
                                                style: (LuthierStyle styleNamed: 'Codigo'))]).
```

7 Uma Ferramenta para Análise de Frameworks

O projeto de Luthier, descrito no capítulo anterior, é o resultado de várias iterações do projeto de um protótipo de visualização para apoio à compreensão de frameworks, desenvolvido experimentalmente durante os últimos dois anos. Esta ferramenta fornece suporte as atividades e requisitos definidos no capítulo quatro, isto é, análise de exemplos de aplicações através de uma meta-arquitetura baseada em um gerenciador de meta-objetos especializado; representação da informação em termos do modelo de hipertexto; diferentes visualizações com manipulação direta; *zoom* semântico baseado em escalas de abstração; seleção e filtragem iterativa da informação a ser visualizada; navegação entre diferentes representações; derivação de abstrações, como subsistemas e padrões de projeto; e construção de livros d documentação.

Este capítulo apresenta os aspectos principais deste protótipo utilizando o próprio Luthier, para mostrar as diferentes funcionalidades providas para análise de frameworks. Em primeiro lugar, na seção 7.1, descreve-se o mecanismo geral e a interface com o usuário provida pela ferramenta para gerenciar a análise de aplicações. A seguir, descreve-se em detalhe a notação de grafos de fluxo de mensagens e sua utilização para a análise detalhada do funcionamento global do framework. Nas seções 7.3 e 7.4, descrevem-se o mecanismo de seleção e filtragem de informação baseado nas mensagens, através de visualizações abstratas de colaborações entre classes, e os mecanismos de *zoom* e seleção dinâmica de visualizações. Os mecanismos de derivação e visualização de subsistemas e padrões de projeto são descritos na seção 7.5. A seção 7.6 apresenta sinteticamente o suporte de análise dinâmica de instâncias, enquanto, a seção 7.7 apresenta exemplos da construção de livros de documentação e seu relacionamento com as ferramentas de visualização[1].

7.1 Interface Geral

O mecanismo básico para a análise de uma aplicação envolve três passos, habitualmente iterativos. Em primeiro lugar, é necessário refletir as classes da aplicação a ser analisada. Após este processo, executa-se a aplicação e os meta-objetos associados no processo anterior, coletam a informação da execução, criando a representação da estrutura do framework utilizando o gerenciador de hipertexto. Se a aplicação é interativa, o usuário pode abrir os diferentes browsers providos para visualizar a informação coletada em um determinado ponto da execução do exemplo do seu interesse. Se a aplicação não é interativa, como por exemplo um compilador, é necessário aguardar até o final da execução, ou suspender sua execução.

A FIGURA 7.1, apresenta a interface com o usuário provida para o gerenciamento de análise de aplicações e da informação coletada. A informação se organiza em bibliotecas que contém a informação relativa a um determinado framework, as aplicações analisadas, as classes correspondentes e os livros associados.

Através desta interface o usuário deve criar uma biblioteca para o framework a ser estudado e depois criar o livro que conterá a informação coletada do framework. Uma vez criada esta estrutura, é necessário definir as aplicações a serem analisadas. Neste processo, o usuário ingressa o nome da classe que representa a aplicação, e a interface apresenta uma lista das potenciais classes que deveriam ser analisadas, através de um algoritmo que constrói uma clausura com as classes que são referenciadas desde essa classe, evitando incluir classes que são parte do sistema Smalltalk base (exceto que o próprio sistema seja analisado). Este conjunto de classes é aproximado devido à ausência de tipos de Smalltalk, mas o suficiente para permitir ao usuário selecionar as principais classes candidatas.

---

[1] Algumas das imagens utilizadas para os exemplos foram editadas para realçar as cores em apresentações complexas.

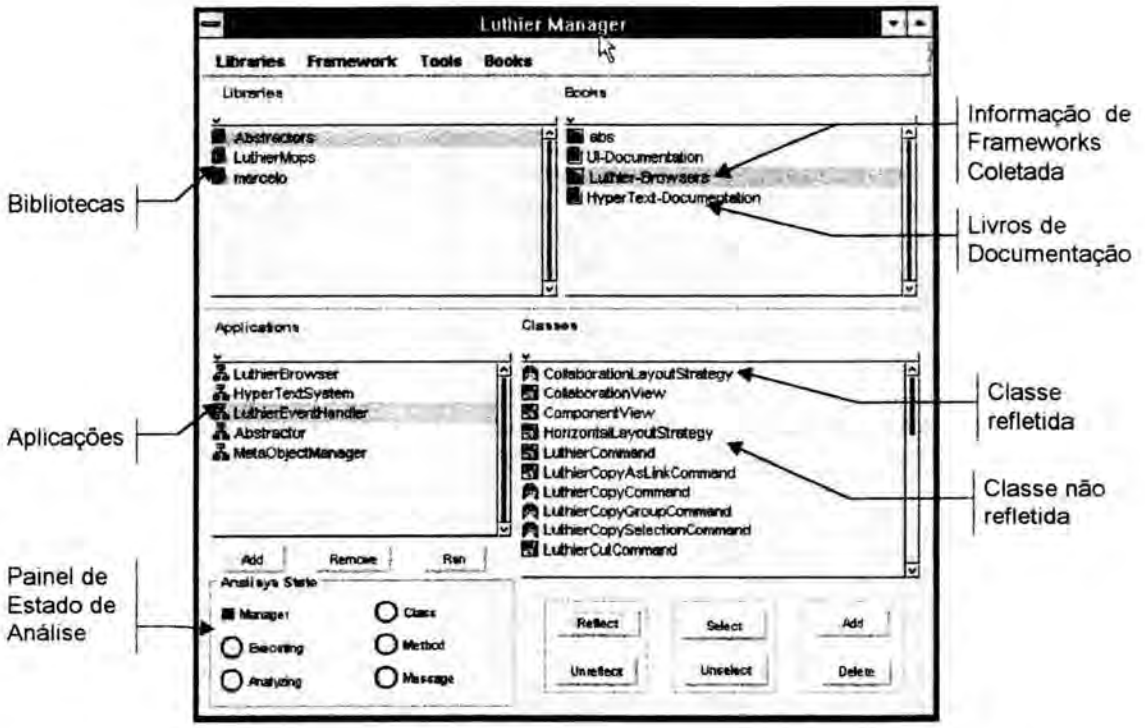


FIGURA 7.1- Interface de gerenciamento de bibliotecas e reflexão de aplicações

Uma vez definido o conjunto de classes, todas ou algumas destas classes, podem ser refletidas para prepará-las para a análise. Iterativamente, novas classes e aplicações podem ser acrescentadas. No exemplo, aparecem cinco aplicações, entre a quais dividiram-se as classes que cumprem com funcionalidades especificas.

Quando uma aplicação é executada, os meta-objetos e o gerenciador associado informam à interface o estado corrente da análise. O painel inferior esquerdo contém botões que mostram dinamicamente o estado corrente de análise, ou seja, execução de código da aplicação, análise de informação e detecção de novas classes, métodos e mensagens não ativados previamente. O botão rotulado *Manager*, permite suspender ou ativar, interativamente, a reflexão de mensagens pelo gerenciador.

A seguir descrevem-se as principais funcionalidades providas pela ferramenta através de exemplos da sua utilização na análise de Luthier.

7.2 Grafos de Fluxo de Mensagens

A notação de grafos de fluxo de mensagens, introduzida no capítulo 4, foi projetada para permitir analisar em forma detalhada os aspectos essenciais de um framework, ou seja, componentes abstratos, categorias de métodos e fluxo de controle de classes e instâncias.

7.2.1 Visualização de componentes abstratos e categorias de métodos

A representação visual de uma classes apresenta a interface dividida em quatro grupos que correspondem com as quatro categorias de métodos já descritas, isto é, uma parte abstrata que agrupa os métodos abstratos, a uma parte que agrupa os método *hook*, uma parte genérica que agrupa os métodos template, e finalmente uma parte que agrupa os métodos base. Esta representação permite identificar rapidamente quais métodos devem ser implementados por subclasses e quais os métodos que definem a estrutura de controle genérica.

A representação também permite visualizar em conjunto o comportamento da classe (definido pelos métodos de classe) e o comportamento das instâncias (definido pelos métodos de instância). Os métodos de classe são mostrados sublinhados para diferenciá-los visualmente.

O componente *Abstractor* (FIGURA 7.2) ilustra a divisão dos quatro tipos de métodos: *abstractionRepresented*: é um método de classe abstrato, *new* é um método de classe do tipo *hook*. *configureAbstrator*: é um método de instância *template* e *interactiveAbstractionLevelChanged* é um método de instância base.

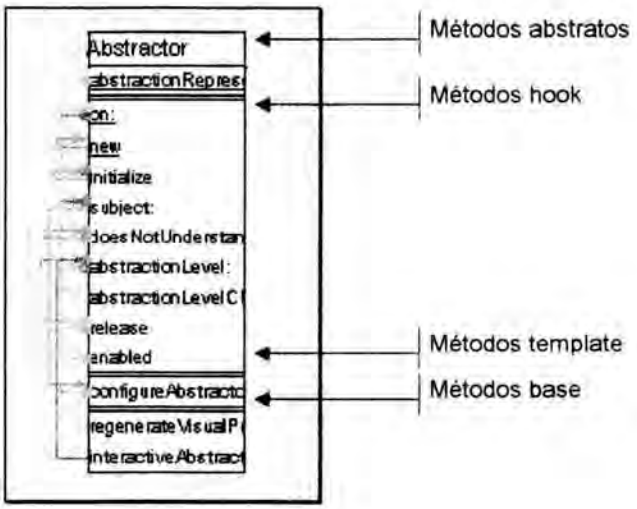


FIGURA 7.2- Representação de uma classe e fluxo de controle interno

Cada classe define um componente que agrupa visualmente a suas subclasses diretas. A apresentação de cada subclasse mostra somente aqueles métodos próprios já que os herdados são mostrados pela representação da sua superclasse. Deste modo é possível visualizar simultaneamente o comportamento genérico do framework, o comportamento próprio das especializações e o fluxo de controle dentro da hierarquia[2].

A FIGURA 7.3 ilustra a hierarquia do componente abstrato *Abstractor* (num estado da execução de um exemplo), na qual aparece o comportamento próprio da classe e os sub-componentes de primeiro nível *KnowledgeAbstratctor*, *ClassAbstractor* e *MethodAbstractor*. *ClassAbstractor*, por sua vez, tem um sub-componente *FrameworkAbstractor* do qual herdam *StructureFrameworkAbstractor* e *SubsystemFrameworkAbstractor*. Também é possível observar o fluxo de controle interno da hierarquia, o qual indica mensagens enviados a *self* ou *super* (em terminologia Smalltalk).

O fluxo de mensagens *inter* e *intra* componentes é representado por setas que unem o método que envia a mensagem e o método ativado pela mensagem. Um circulo na ponta, e/ou origem da seta, indica que a mensagem é trocada entre diferentes instâncias. Esta informação é valiosa para identificar a funcionalidade do relacionamento definido pela mensagem, ou seja, 1:1, 1:N ou M:N.

A diferenciação entre métodos de classe e instância é importante para identificar a seqüência de criação de instâncias de cada componente. A criação de instâncias geralmente envolve seqüências de inicialização que devem ser respeitadas pelas subclasses.

[2] Esta estratégia de visualização pode ser modificada para mostrar todos os métodos definidos em cada componente, caso isto seja necessário.

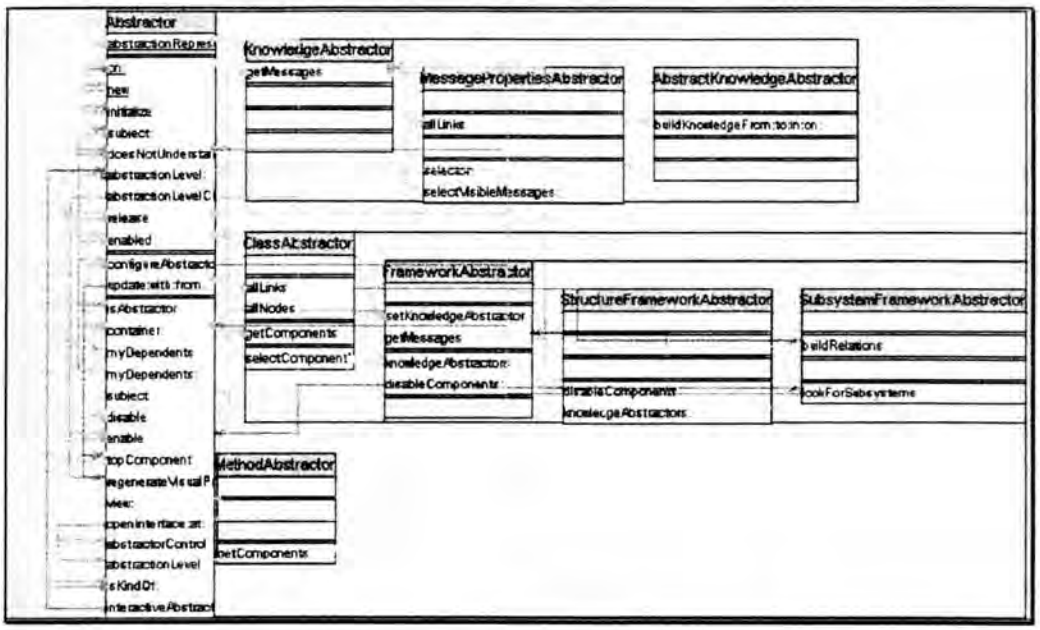


FIGURA 7.3. Representação de hierarquia de classes e fluxo de controle interno

A informação das mensagens combinada com os tipos de métodos permite condensar, na apresentação do componente abstrato, uma grande quantidade de informação relativa a extensão de métodos *hook* e a forma que as instâncias são organizadas.

A cor das setas sugere a ordem relativa na seqüência de mensagens. Cores perto do azul indicam fluxos iniciais, seguidos por fluxos verdes, amarelos, laranja até vermelhos. A FIGURA 7.4 mostra o comportamento global entre os componentes de interface com o usuário, os abstratores e os componentes. A codificação de cores sugere que as interações mostradas começam nas visões, as quais interagem com abstratores e com manejadores de eventos. No primeiro caso a seqüência de controle passa para os componentes do modelo de hipertexto e no segundo para os comandos, tal com é a seqüência normal de comunicação do framework descrita no capítulo anterior.

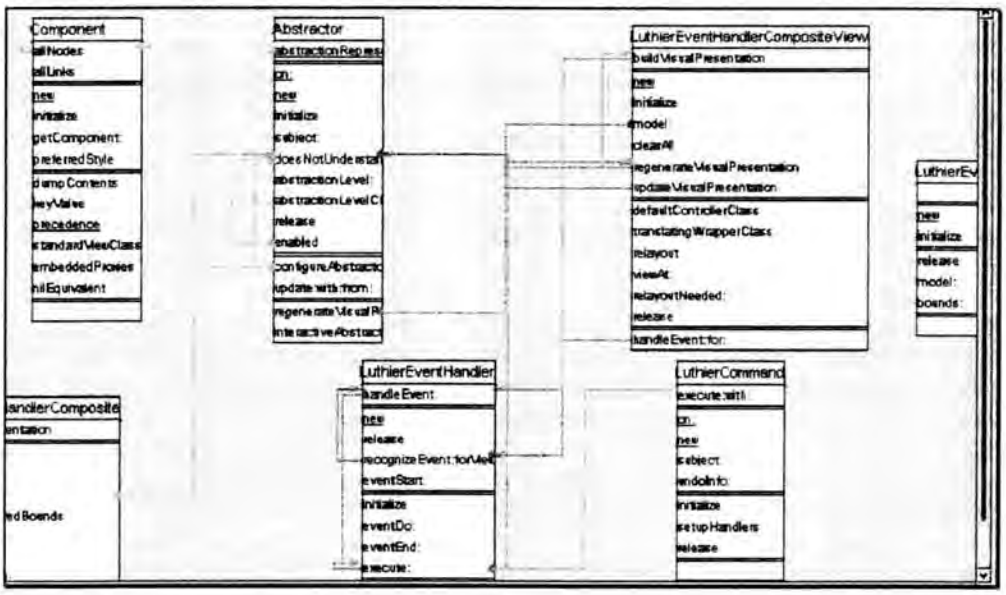


FIGURA 7.4- Fluxo de controle entre diferentes componentes

7.2.2 Análise detalhada da estrutura de controle

A representação gráfica apresenta o problema de saturação visual produzida pela quantidade de elos, quando as interações são numerosas, porém é de utilidade para visualizar a complexidade global da arquitetura e os pontos nos quais existe maior densidade de troca de mensagens. Assim, para realizar a análise detalhada das interações a ferramenta provê mecanismos para filtrar a visualização e navegação que facilitam a análise detalhada do comportamento de seqüências de controle específicas.

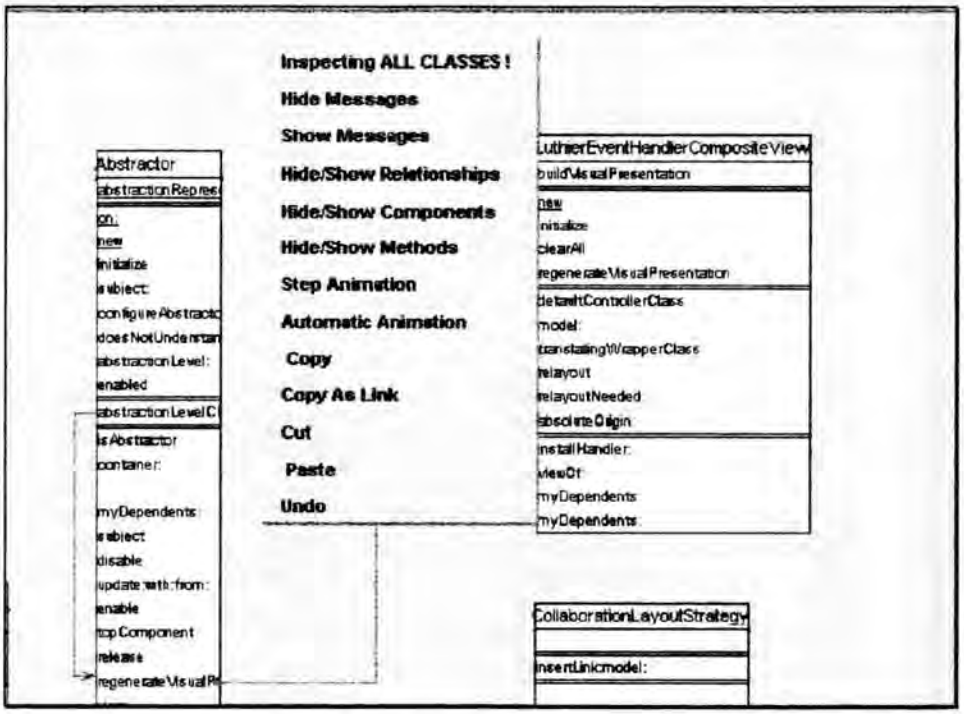


FIGURA 7.5- Opções de visualização

A FIGURA 7.5 apresenta um exemplo das capacidades fornecidas pela interface para visualizar as mensagens recebidas e enviadas por um método. O usuário pode solicitar visualizar quais são as mensagens que ativam um método e realizar o seguimento destas mensagens em sentido inverso. Também é possível visualizar as mensagens enviadas por um método e realizar o seguimento normal do fluxo das mensagens. Isto permite realizar uma análise detalhada de como os componentes interagem, focalizando a atenção nos pontos de maior interesse. Esta funcionalidade é de grande utilidade para determinar qual foi o fluxo de mensagens que produziu a ativação de um dado método. No exemplo da figura, foram ocultadas todas as mensagens e se solicitou mostrar as mensagens que são enviadas e que envia o método *regenererateVisualPresentation* da classe *Abstractor*.

Além disso, é possível acessar o código que implementa um método e a todas suas redefinições na hierarquia, permitindo assim analisar nos contextos nos quais um método é ativado (FIGURA 7.6).

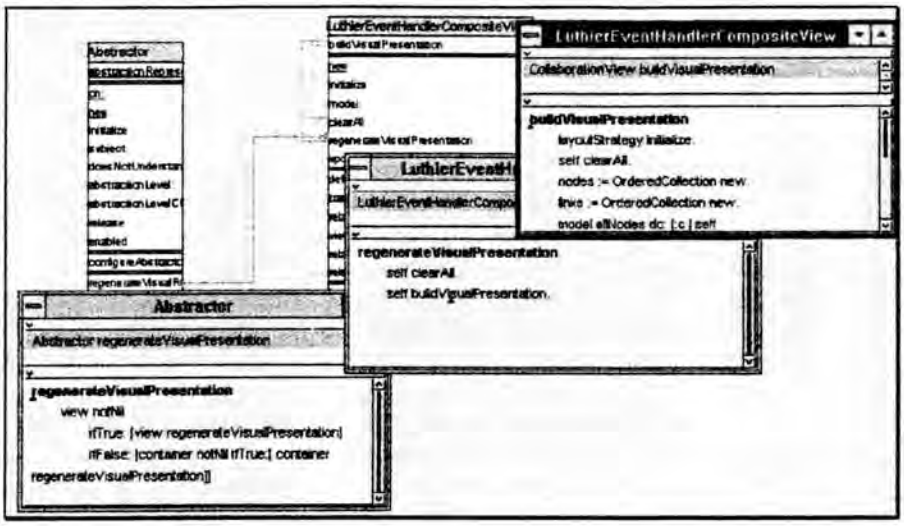


FIGURA 7.6- Inspeção de código de métodos

7.2.2.1 Animação arquitetônica

A seqüência de envio de mensagens é importante para compreender a dinâmica das interações, e fundamentalmente os caminhos de controle codificados pelo framework. Para isto, a interface com o usuário suporta a animação da seqüência de passagem de mensagens. O seguimento pode ser global, ou seja, o seguimento de toda a execução, ou local mostrando só a seqüência de mensagens enviadas por um método determinado. Nesta animação, o cursor se desloca na tela indicando o método chamado, e cores são utilizadas para ressaltar qual a classe da instância que recebe a mensagem e qual a classe que implementa essa mensagem. Só são mostrados passos de controle resumidos, não todas as mensagens, o qual permite realizar o seguimento do comportamento da arquitetura sem redundância. Cada passo é visualizado só uma vez permitindo ao usuário concentrar-se no funcionamento global da arquitetura, pois evita o seguimento de ciclos desnecessários.

A FIGURA 7.7 apresenta uma imagem da ferramenta na qual o usuário está realizando a animação da seqüência de mensagens entre a classe *LuthierBookAccessor* e *Component*.

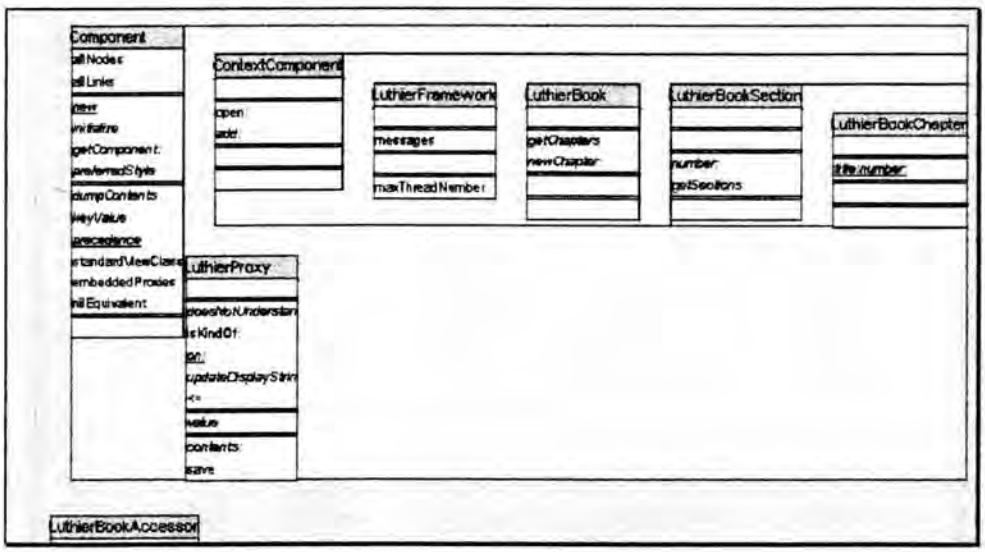


FIGURA 7.7- Passo na animação da seqüência de mensagens

Nesse ponto da execução a classe *LuthierProxy* é ressaltada como a classe receptora e implementadora da mensagem *dumpContents*, o qual é um método *hook* definido por *Component*, redefinido por *LuthierProxy*.

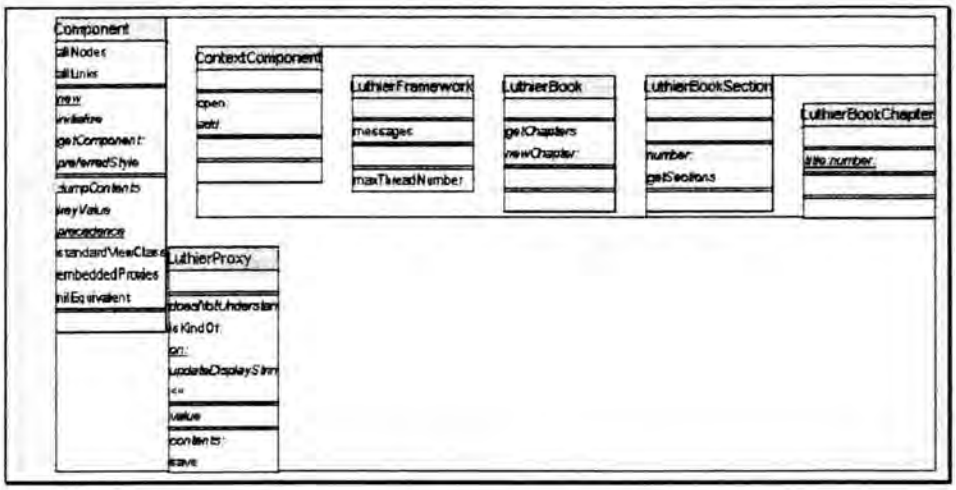


FIGURA 7.8- Propagação de mensagem

No passo seguinte (FIGURA 7.8) o método *dumpContents* aparece como sendo chamado novamente, mas agora o receptor é uma instância de *LuthierBook*, o qual indica que um *proxy* está propagando a mensagem para seu componente.

A FIGURA 7.9 apresenta o passo seguinte na execução do método *dumpContents*, a implementação provida por *LuthierBook* envia a mensagem *embeddedProxies*, a uma instância de *LuthierBookChapter*, sendo ativada a implementação herdada da classe *LuthierBookSection*. Desta forma é possível observar simultaneamente o relacionamento entre instâncias, bem como as classes nas quais os métodos do framework são redefinidos.

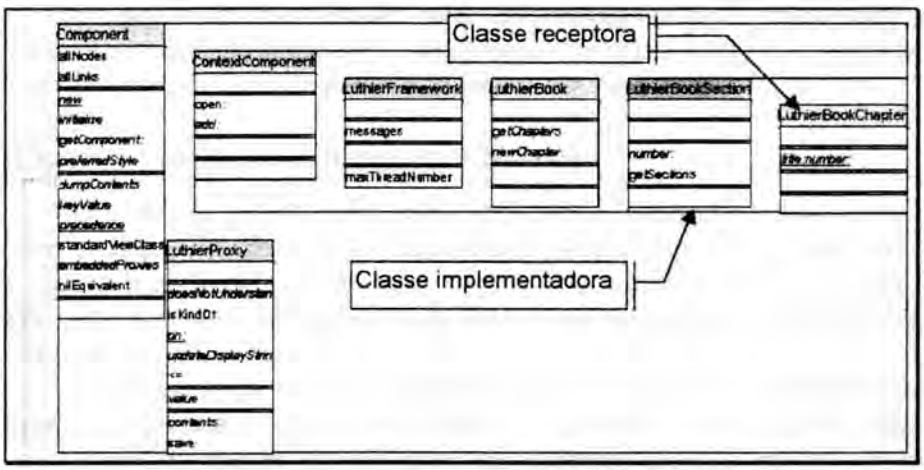


FIGURA 7.9- Ativação de método herdado

No passo seguinte, mostrado na FIGURA 7.10, uma instância de *LuthierProxy* recebe a mensagem *nilEquivalent*, enviada por um *LuthierBookChapter*. Neste ponto começa a se observar como as instâncias das classes analisadas tem um *proxy* associado, o qual propaga as mensagens que não trata diretamente. O fato do método *dumpContents* ser propagado entre diferentes instâncias dentro da hierarquia, sugere que os objetos estão compostos recursivamente, e, existindo diferentes redefinições

do método, é possivel deduzir que um livro, contem capitulos os quais contém *proxies* a outros elementos.

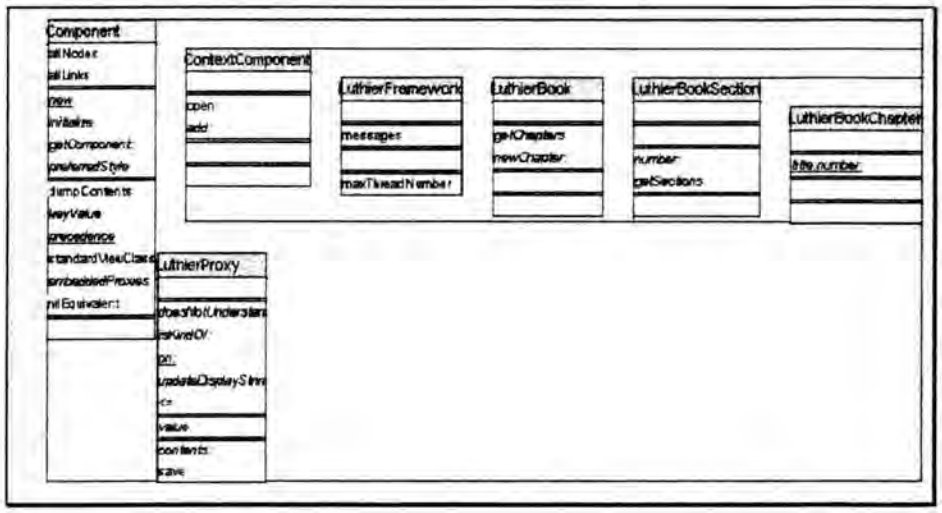


FIGURA 7.10- Retorno de controle para os *proxies*

A visualização única dos métodos de uma hierarquia, em conjunto com a capacidade de animação, facilita o reconhecimento visual do fluxo de controle próprio do framework. A seqüência de mensagens que tem um caminho de controle comum permanece inalterada na tela, enquanto se animam os caminhos de controle que cada especialização implementa.

O mecanismo de animação descrito é implementado através de comandos. Quando o usuário solicita a animação desde um dado método, uma instância de comando que implementa essa animação é criada. Este comando instala na visão de nível superior um manejador do evento *click* para controlar cada passo da animação e um de doble *click* para interromper animação. Estes manejadores são removidos quando a animação é finalizada.

Esta implementação permite que diferentes tipos de animações possam ser acrescentadas sem necessidade de modificar a visualização, como por exemplo, o seguimento completo de todas as mensagens trocadas durante a execução analisada.

7.3 Visualização Abstrata, Filtragem e Seleção

A visualização provida pelos grafos de fluxo de mensagens é de utilidade para analisar o comportamento detalhado da estrutura de controle provida pelo framework, mas, não é totalmente adequada como ponto de inicio para compreender, em um nivel global, as interações tanto entre classes abstratas quanto entre suas especializações. Deste modo, uma visualização mais abstrata, tal como a apresentada na FIGURA 7.11, é necessária.

A figura apresenta um visão global do browser de relacionamentos entre classes, no nível de relacionamentos entre as classes abstratas do framework. Nesta representação, triângulos representam hierarquias de classes e retângulos representam classes simples. As setas simples representam relacionamentos cliente-servidor, enquanto as setas duplas indicam um relacionamento cliente-servidor mútuo entre duas classes.

Os paneis inferiores contêm componentes de interface que habilitam ao usuário a selecionar qual a informação e atributos que devem ser mostrados pela visualização, e manipular interativamente a escala de abstração para definir o grau de detalhe da visualização.

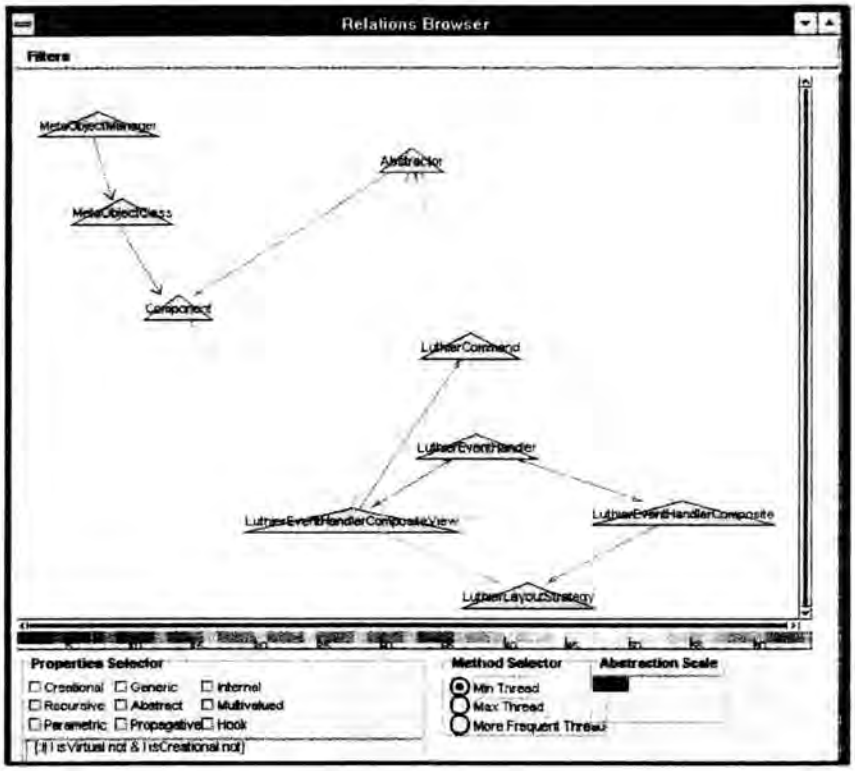


FIGURA 7.11- Visualização de relacionamentos entre classes abstratas

A cor dos relacionamentos dá uma noção relativa da ordem no tempo na qual esses relacionamentos se estabelecem através da troca de alguma mensagem. Os relacionamentos mostrados neste nível representam conjuntos de mensagens, assim, através da interface de seletores de atributos, a semântica da cor utilizada pode ser variada interativamente para representar a ordem relativa dentro de um *thread* [3] de acordo a três critérios:

- ordem da primeira mensagem trocada entre duas classes
- ordem da última mensagem trocada entre duas classes
- ordem da mensagem mais freqüentemente trocada entre duas classes

Quando o usuário seleciona um botão deste painel, a visualização é automaticamente atualizada, variando a cor das setas de acordo com a opção selecionada. A FIGURA 7.11 apresenta a visão da seleção de cores pela primeira mensagem em ordem dentro de um thread de execução. As cores perto do azul escuro mostram relacionamentos produzidos nos começos da execução, como é o caso dos relacionamentos entre *MetaObjectManager*, *MetaObjectClass* e *Component*, os quais indicam troca de mensagens entre subclasses destas classes durante a análise da aplicação. Já os relacionamentos mais claros indicam seqüências que se iniciaram mais tarde, como é o caso da seqüência entre *LuthierEventHandlerView*, *LuthierEventHandler* e *LuthierCommand*, a qual corresponde com processo de tratamento de eventos iniciado sempre por uma visão.

A FIGURA 7.12 apresenta as duas visões complementares, resultantes de selecionar o maior thread e o mais freqüente. Pode-se observar que em ambos casos a seqüência de relacionamentos entre as classes sugere uma ordem que se inicia nas visões, continua pelos manejadores de eventos, os

[3] Utiliza-se o termo *thread* para referenciar genericamente um caminho de controle dentro do fluxo de controle total do framework, não indicando, necessariamente, uma execução concorrente desses caminhos.

comandos, abstratores e finalmente os componentes.

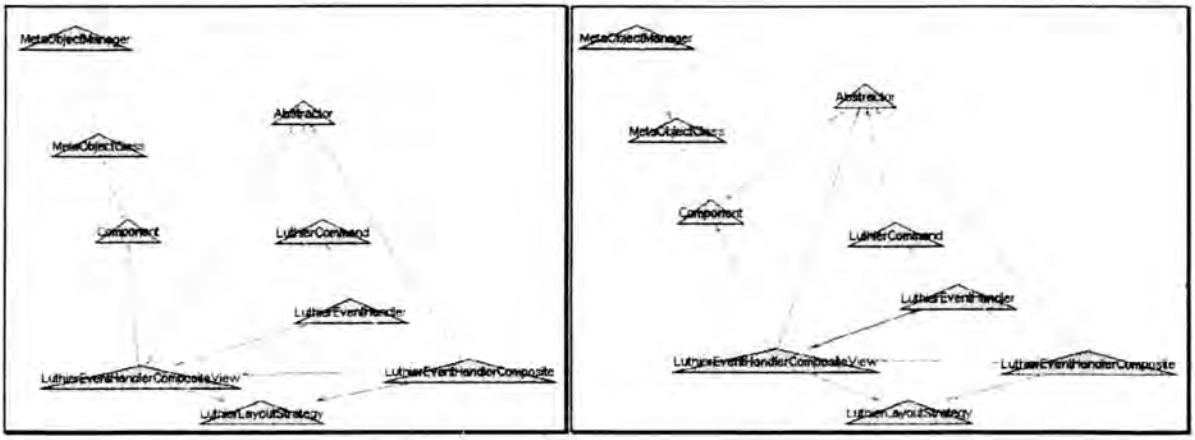


FIGURA 7.12- Seleções por maior e mais freqüente *thread*

O nível de detalhe desta representação é governado por abstratores, segundo a escala:

(abstractHierarchy topLevelHierarchy concreteHierachy)

A posição corrente nesta escala pode ser governada interativamente pelo usuário, através da barra de deslizamento mostrada no painel inferior direito da FIGURA 7.11. A FIGURA 7.13 mostra a visualização resultante de selecionar na interface da escala de abstração, o nível seguinte dessa escala (*topLevelHierarchy*), o qual permite a visualização dos topos das hierarquias não abstratos, como por exemplo *LuthierBrowser*, bem como classes que não possuem subclasses, como *LuthierEvent* por exemplo.

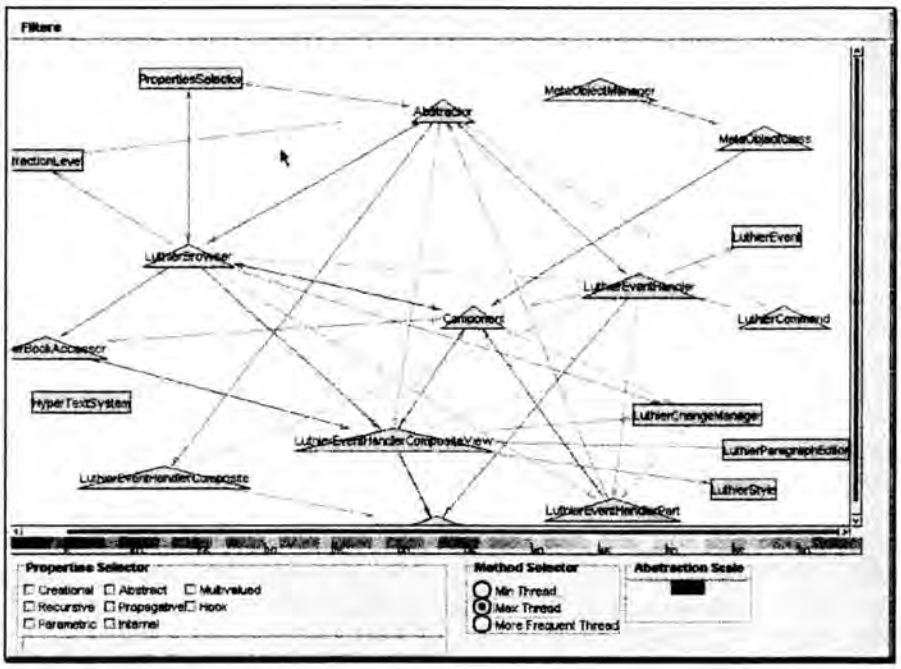


FIGURA 7.13- Visualização de relacionamentos a nível de topos de hierarquias

7.3.1 Filtragem e seleção baseada nas mensagens

Além da informação provida pelas cores dos relacionamentos, também é importante dispor de mecanismos que permitam visualizar aspectos parciais da informação, como por exemplo centrar a atenção em classes que cumprem com determinadas propriedades relevantes.

Mecanismos centrados na seleção de mensagens que satisfazem determinadas propriedades, oferecem a possibilidade de explorar a estrutura do framework no nivel abstrato, para procurar estruturas relevantes, como por exemplo cadeias de propagação de mensagens ou fluxos de criação de instâncias. Para isto, um mecanismo de seleção baseado em propriedades das mensagems é provido.

Uma *propriedade de mensagem* identifica ou nomeia um *contexto predeterminado* no qual uma dada seqüência de mensagens é enviada e recebida. Sob esta abordagem mensagens enviadas são identificadas por uma combinação de propriedades, as quais podem ser utilizadas posteriormente para filtrar a visualização da arquitetura de acordo com as mensagens que cumprem com certa combinação de propriedades.

Dependendo do contexto no qual mensagens são enviadas, elas podem ser identificadas segundo a lista preliminar de propriedades não mutuamente excludentes:

- ***Abstrato:*** Identifica mensagens que ativam métodos que provem a implementação de métodos abstratos definidos em classes abstratas. Este tipo de mensagens identificam o fluxo de controle abstrato do framework.
- ***Extensional:*** Esta categoria identifica mensagens que ativam métodos *hook*, isto é, métodos concretos que são redefinidos dentro da hierarquia de classes. Estes mensagens identificam pontos de extensão dos serviços providos pelo framework.
- ***Propagativo:*** Esta propriedade identifica mensagens que são propagados através de diferentes instâncias. Estes tipo de mensagens caraterizam cadeias de responsabilidades e delegação de responsabilidades.
- ***Recursivo:*** Estes são mensagens propagados a componentes pertencentes à mesma hierarquia de classes, isto é, classes com um ancestro comum. Eles caraterizam o comportamento de objetos compostos recursivamente.
- ***Criacional*:** Esta propriedade identifica mensagens que pertencem a uma seqüência iniciada por um método de criação de instâncias ou enviadas a um método criacional. Estas seqüências são úteis para identificar seqüências de instanciação do framework e protocolos de inicialização de componentes. Também são úteis para identificar quando instâncias são criadas para satisfazer um requerimento específico.
- ***Paramétrico*:** Mensagens enviados a instâncias que são retornadas por uma mensagem ou recebidas como argumento, isto é, mensagens enviadas a instâncias não diretamente referenciadas através de variáveis de instância do objeto que envia a mensagem. Este propriedade identifica fluxo de controle sobre instâncias que são mantidas por objetos servidores e fluxo de controle sobre instâncias que fluem como parâmetros através de uma rede de objetos..
- ***Multivalorado*:** Mensagens que são enviados a diferentes instâncias, denotando relacionamentos 1:N ou N:N. Esta propriedade é útil para compreender a funcionalidade dos relacionamentos entre classes e a composição interna de objetos.
- ***Interno*:** Mensagens que ativam métodos implementados ou herdados por uma classe. Eles identificam o fluxo de controle interno que ativa métodos base ou a extensão de funcionalidade do framework através de subclasses.

Através destas propriedades um usuário pode filtrar qualquer visualização para mostrar só as classes e métodos relacionados por mensagens que satisfazem uma dada combinação de propriedades. Por exemplo, fluxos que satisfazem a condição de serem (*extensional*| *abstrato*) & *multivalorados*, são úteis para filtrar a visualização de acordo a uma perspectiva de meta-padrões [PRE 94]. Neste caso, métodos *template* e métodos *hook* relacionados serão selecionados, permitindo visualizar os pontos de extensão do framework.

Desde uma perspectiva de padrões de projeto, a análise de fluxos recursivos permite ao usuário buscar pela existência de alguma materialização dos padrões compostos, como *Composite* ou *Decorator*. Fluxos *propagativos* e *multivalorados* podem também sugerir a existência de um padrão *Chain of Responsibility*, isto é, uma seqüência da mesma mensagem enviada a diferentes instâncias com diferente freqüência. Uma análise detalhada do código e das instâncias envolvidas podem ajudar a determinar se o comportamento codificado pelas classes corresponde com alguns destes padrões.

A interface de seleção de propriedades mostrada no painel inferior esquerdo da FIGURA 7.11, permite a seleção direta destas propriedades, bem como o ingresso textual de outras consultas desejadas pelo usuário. Quando uma propriedade é selecionada ou uma consulta ingressada, a visualização se reconfigura automaticamente para mostrar só aqueles elementos que satisfazem a consulta dada.

Por exemplo, a FIGURA 7.14 apresenta o resultado de filtrar a visualização da FIGURA 7.13 pelas mensagens que cumprem com a propriedade de serem genéricas, isto é, mensagens que ativam métodos abstratos, *hook* ou *templates*. Como é possivel observar, a visualização é transformada para mostrar só as classes que são relacionadas por mensagens que cumprem com essas propriedades. O mecanismo de seleção é implementado por um abstrator, o qual é o encarregado de selecionar as mensagens de acordo com o critério estabelecido e habilitar aqueles abstratores relacionados com classes e métodos que são origem ou destino de cada mensagem selecionada.

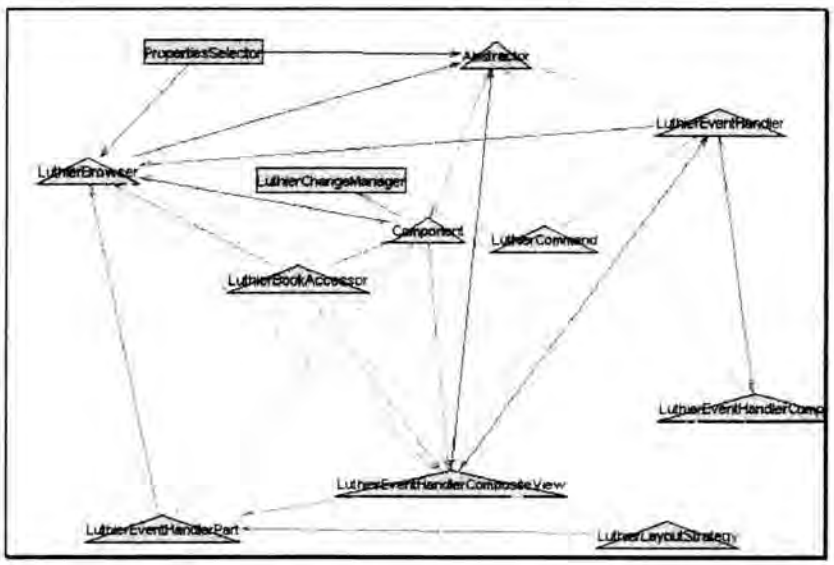


FIGURA 7.14- Filtragem para visualização dos relacionamentos genéricos

Alternativamente, o usuário pode ingressar consultas textuais, na forma de um bloco de código Smalltalk, para selecionar visualizar só um determinado conjunto de classes e, seus correspondentes relacionamentos, também baseado nas seleção de mensagens. Por exemplo, o bloco:

```
[:m| m originClass = LuthierProxy | m targetClass = LuthierProxy]
```

especifica a seleção de aquelas mensagens que são enviadas ou recebidas por instâncias da classe

LuthierProxy. A FIGURA 7.15, apresenta o resultado da visualização original filtrada para mostrar só aquelas classes relacionadas de acordo com esse bloco. Esta visualização pode, por sua vez, ser variada no nível de detalhe, mostrando as interação representadas a nível de componentes abstratos, nível de hierarquia e concretos.

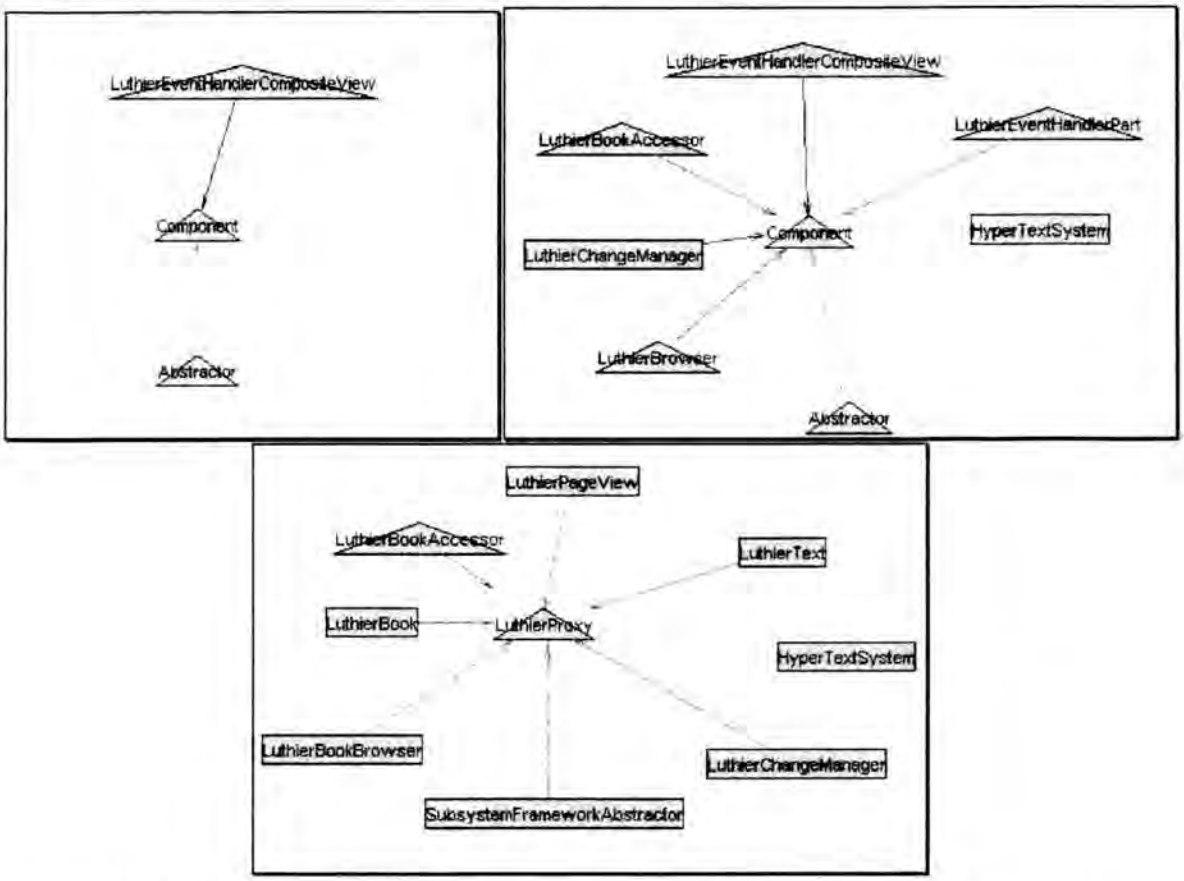


FIGURA 7.15- Diferentes níveis de visualização para a seleção de classes que interagem com *LuthierProxy*

7.3.2 Aspectos de implementação

É interessante destacar aqui, a forma na qual os abstratores e seletores podem ser compostos para dotar um browser com diferentes capacidades. O browser de relacionamentos é configurado através da implementação do método mostrada a seguir:

```
configureBrowserFor: aContext
    selector := PropertiesSelector new properties: #(#Creational #Hook #Abstract #Internal
                                                     #Propagative #Recursive #Multivalued
                                                     #Parametric #Generic).
    valueSelector := (ExclusiveSelectorCollection new)
                        add: (ValueEvaluator
                                attribute: #initialThread
                                function: #min:
                                name: 'Min Thread');
                        add: (ValueEvaluator
                                attribute: #initialThread
                                function: #max:
                                name: 'Max Thread');
                        add: (ValueSelector
                                attribute: #timesActivated
                                function: #>
```

```
                                        name: 'More Frequent Thread'
                                        valueSelector: #initialThread).

valueSelector forClass: LuthierKnowledgeRelation onMethod: #initialThread.

abstractor := FrameworkAbstractor on: ( (MessagePropertiesAbstractor on: aContext )
                                                    selector:selector).
abstractor abstractionLevel:(AbstractionLevel scale:#( #abstractHierarchy #
                                                topLevelHierarchy #concreteHierarchy )).

topView := RelationsView new.
topView model: abstractor.
```

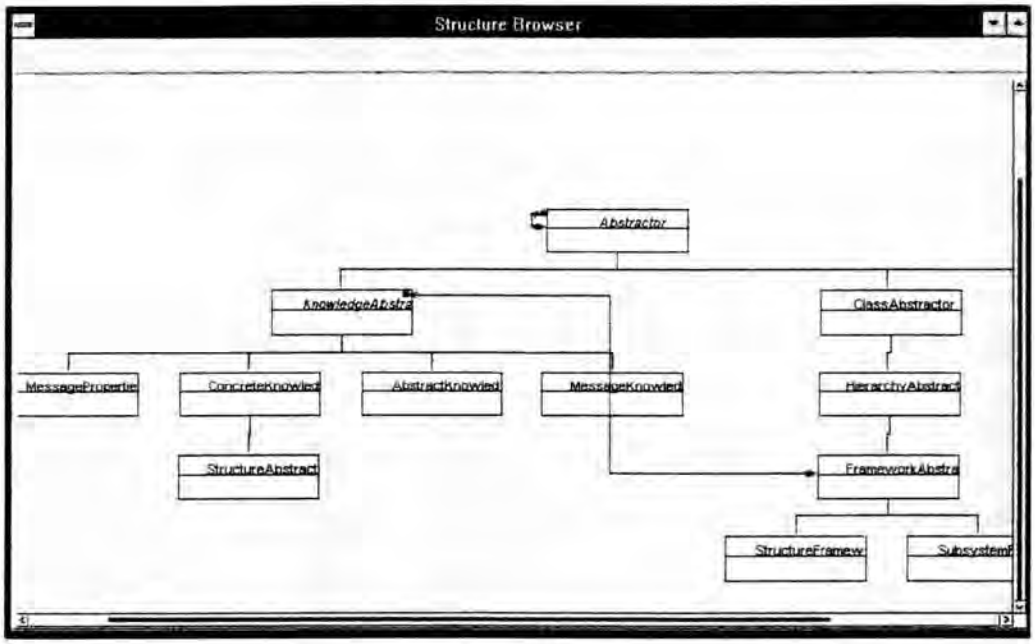


FIGURA 7.16- Diagrama OMT da hierarquia de abstratores utilizada pelos browsers

Neste método o browser é configurado para criar uma visualização que tem como modelo um abstrator de frameworks, o qual gerência a seleção de algoritmos que geram os relacionamentos abstratos e concretos, e instala recursivamente abstratores de hierarquias sobre as hierarquias de classes componentes, estes por sua vez instalam abstratores de classes e, finalmente estes últimos instalam abstratores de métodos, criando assim a hierarquia de abstratores. A FIGURA 7.16 apresenta o diagrama estático OMT da hierarquia de abstratores, gerado pela visualização de estruturas provida pela ferramenta. No diagrama podem se observar dois relacionamentos de composição que forma derivados das caraterísticas das mensagens trocadas entre instâncias de abstratores: O relacionamento definido pela classe Abstractor para conter a hierarquia de abstratores e o relacionamento especializado de FrameworkAbstractor com KnowledgeAbstractor, que denota o relacionamento entre o primeiro e os abstratores que representam os relacionamentos abstratos, concretos e de mensagens.

O abstrator de frameworks é instalado sobre o abstrator encarregado da seleção de mensagens, *MessagePropertiesAbstractor*, o qual, redefine o protocolo padrão para aplicar o bloco de seleção a cada mensagem retornada pelo seu sujeito, quando as visualizações solicitam os elos contidos no modelo. Este abstrator recebe também como argumento um instância do seletor que será utilizado para ingressar as consultas.

Os seletores provêm um mecanismo adaptável para especificar critérios para selecionar ou avaliar um dado atributo de um conjunto de objetos. A FIGURA 7.17 apresenta o esquema de

ativação dos seletores instalados no exemplo. A classe *ExclusiveSelectorCollection* fornece o implementação da seleção da semântica das cores dos relacionamentos descrita acima. Para isto utiliza-se do suporte de meta-objetos para interceptar uma dada mensagem, através da qual as visões solicitam o valor do atributo que definirá a cor (*initialThread* no exemplo). Quando uma visão envia a um elo esta mensagem, ela é interceptada por um meta-objeto (*EvaluatorMetaObject*), o qual recupera a coleção de mensagens representados pelo relacionamento e solicita ao avaliador associado que selecione o valor adequado dentre desse conjunto. O seletor composto deriva a seleção para o seletor ou avaliador correntemente selecionado pelo usuário desde a interface, o qual seleciona, no caso da figura, o maior valor do atributo *initialThread* dentre as mensagem representadas pelo elo. Este valor é retornada para a visão como se fosse o valor real do atributo solicitado.

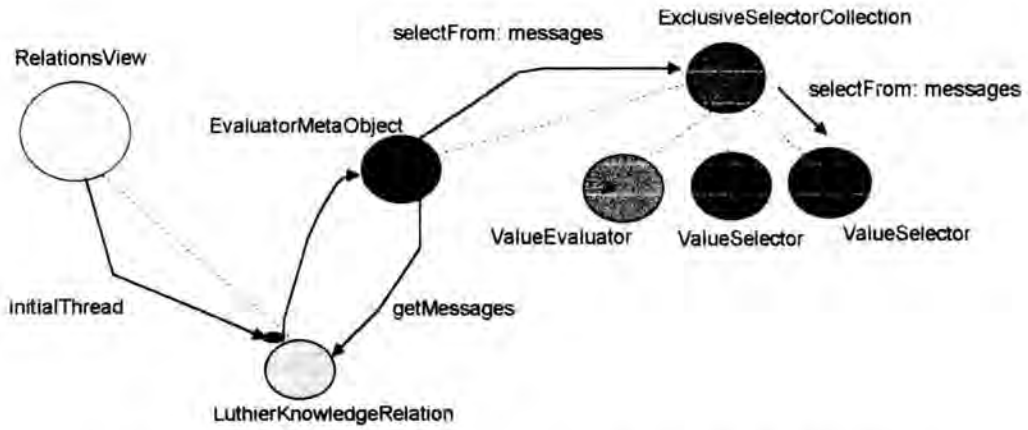


FIGURA 7.17- Esquema do mecanismo de ativação de seletores

Este mecanismo permite que as visões sejam programadas independentemente da semântica atribuida aos atributos visuais, como por exemplo cor, e seja determinada externamente pelo usuário, sem necessidade de programar em cada visualização particular, mecanismos especializados para tal fim.

7.4 *Zooming* entre Representações

Utilizando os mecanismos de filtragem com a visualização abstrata de relacionamentos, é possivel isolar partes do framework que satisfazem determinadas propriedades ou critérios. Para analisar e compreender como esses relacionamentos são estabelecidos, a ferramenta permite realizar a seleção de subconjuntos das classes visualizadas e realizar o *zooming* dessas classes, utilizando visualizações alternativas, como por exemplo grafos de fluxo de mensagens para analisar o fluxo das mensagens que definem os relacionamentos, ou diagramas estáticos OMT para visualizar a conformação das hierarquias e os relacionamentos de composição e associação.

FIGURA 7.18 apresenta um exemplo da visualização estática OMT de três classes selecionadas da representação abstrata, *Component*, *Abstractor* e *LuthierEventHandlerComposite-View*[4], as quais são mostradas no nivel de abstração de classes abstratas. Na figura pode observar-se que os relacionamentos mostrados na visão abstrata são relacionamentos de composição. Estes relacionamentos são deduzidos tomando em consideração que existe composição quando existe propagação de uma mesma mensagem entre diferentes instâncias. O relacionamento recursivo na classe *Abstractor* representa a composição de abstratores já descrita, enquanto o relacionamento com a classe *Component* representa o relacionamento *abstrator-sujeito* também descrito. As visões por sua vez são

[4] O nome desta classe aparece parcialmente na figura pois os componentes foram projetados para ter a mesma largura.

compostas por subvisões, fato representado pelo relacionamento recursivo na classe *LuthierEventHandlerCompositeView*, bem como o relacionamento entre visões e abstratores reflete o fato de que as visões propagam mensagens para os abstratores que são seu modelo. A FIGURA 7.20 apresenta mesma visualização tendo variado o nível da escala de abstração para visualizar a hierarquia de classes concretas. Na figura observa-se a hierarquia definida pela classe *Component*, e um relacionamento de composição entre a classe *LuthierProxy* e *Component*, o qual representa o relacionamento entre os *proxies* es os componentes do modelo de hipertexto quando são feitos persistentes.

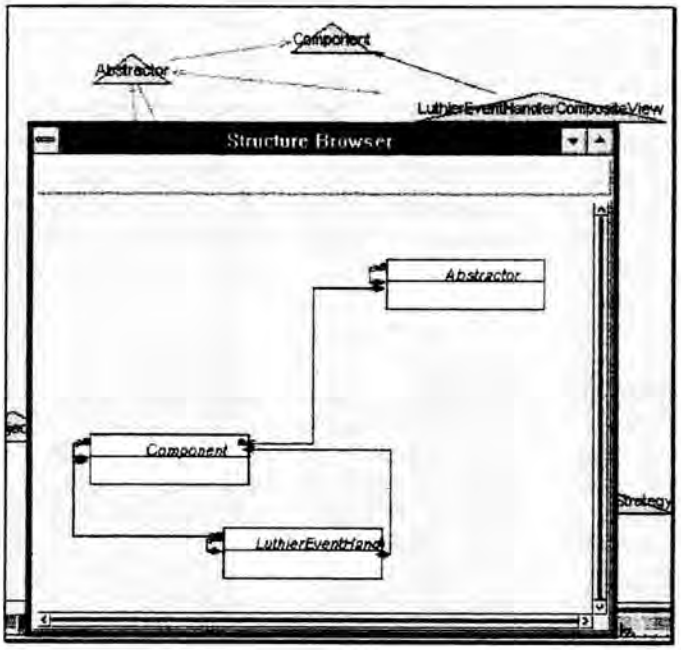


FIGURA 7.18- Zoom dos relacionamentos entre três classes com notação OMT

A FIGURA 7.19 apresenta o *zoom*, realizado sobre a apresentação OMT, utilizando grafos fluxo de mensagens para visualizar as mensagens que definem os relacionamentos. Sobre esta apresentação o usuário pode utilizar os mecanismos de animação, ocultamento e navegação até o código já descritos, para analisar a seqüência em que as mensagens são trocadas e a implementação dos métodos envolvidos.

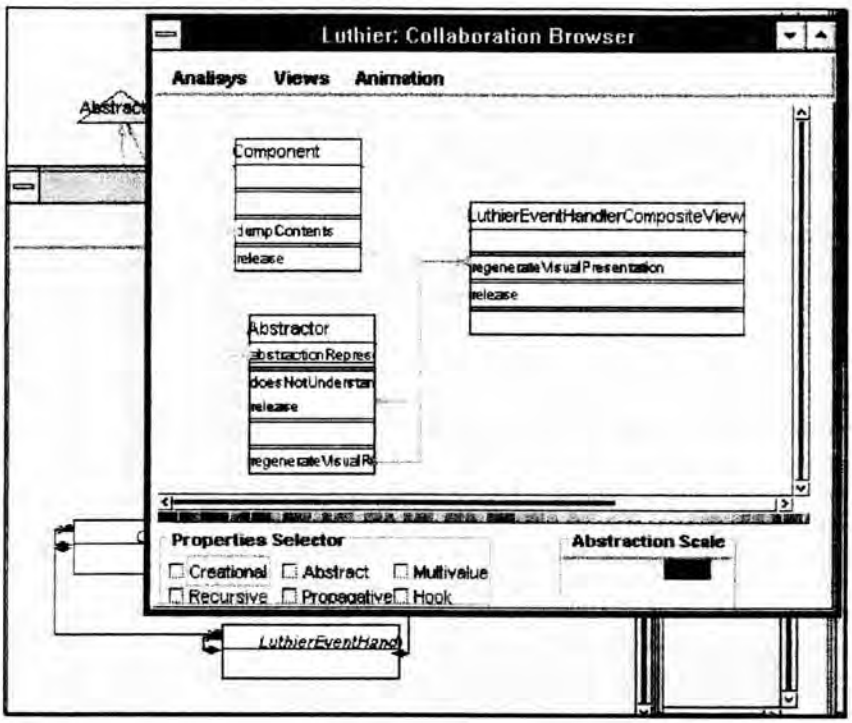


FIGURA 7.19- Zoom da visualização anterior com fluxo de mensagens

Através do mecanismo de *zoom* entre representações, a ferramenta fornece o meio para explorar diferentes porções do framework analisado, desde diferentes perspectivas. Isto permite, uma compreensão gradual, tanto da estrutura estática como do comportamento da arquitetura implementada pelo framework, como seus métodos são redefinidos e como são os relacionamentos principais entre instâncias das classes concretas.

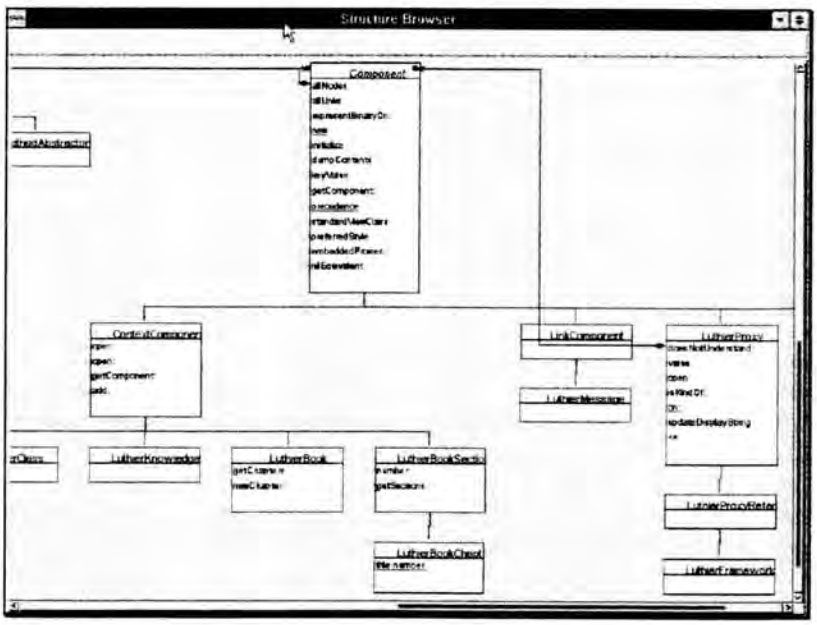


FIGURA 7.20- Visualização no nível de hierarquia concreta da classe *Component*

7.5 Análise de Subsistemas e Padrões de Projeto

O reconhecimento abstrações, que não são explicitamente suportadas pela linguagem de programação, é um complemento importante para facilitar a compreensão global da funcionalidade de um framework. Tal como é o caso de Luthier, muitos frameworks são projetados como um conjunto de sub-frameworks que interagem entre si, os quais fornecem suporte extensível para funcionalidades bem definidas. O reconhecimento destas unidades funcionais é um complemento importante para fornecer ao usuário visões iniciais mais abstratas.

Também, a identificação de potenciais padrões de projeto que possam existir dentro da estrutura do framework é um complemento importante para facilitar a compreensão de como partes do framework foram projetadas e a função que determinados métodos cumprem dentro de uma classe dada.

A seguir descrevem-se o suporte provido por Luthier para a identificação e visualização de ambos tipos de abstrações.

7.5.1 Subsistemas e grupos colaborativos

A identificação de subsistemas ou grupos colaborativos não pode ser efetuada com total exatidão de maneira automática, pois, no caso geral, estes grupos são resultado de um agrupamento que responde a critérios funcionais que não podem ser automaticamente deduzidos das interações entre classes, sem conhecimento adicional. Muitos subsistemas são projetados para fornecer um conjunto de classes que são utilizadas por clientes externos, as quais interagem entre elas, bem como com outras classes que recebem como parâmetros. Assim, um critério baseado, exclusivamente, nas carateristicas topológicas do grafo definido pelas interações entre classes, coletadas pela ferramenta, não permite deduzir com total certeza a composição de um subsistema. Por esta razão, o agrupamento realizado representa uma sugestão da existência de um *potencial* subsistema ou grupo colaborativo.

O browser de subsistemas utiliza um abstrator que encapsula o algoritmo utilizado para criar os grupos de classes e informar estes grupos á visualização, como se realmente fossem parte da informação contida no modelo, quando a visualização solicita os nodos a serem visualizados.

O algoritmo de agrupamento é um algoritmo que constrói clausuras de classes, utilizando uma combinação de heurísticas e análise de interações. No caso geral, classes que implementam funcionalidades relacionadas são nomeadas com nomes semelhantes. Assim, classes que começam ou finalizam com cadeias semelhantes são agrupadas em primeiro lugar para formarem uma primeira aproximação de um subsistema. Após este agrupamento, são incluídas classes que só colaboram, ou tem relacionamento cliente-servidor mútuo, com classes dentro de um dado grupo, e excluídas classes que colaboram com outras pertencentes a grupos diferentes, até que todas as classes tenham sido analisadas.

A FIGURA 7.21 apresenta a visão de Luthier, utilizando notação de Grafo de Colaborações do projeto conduzido por responsabilidades [WIR 90]. O browser de subsistemas, utiliza uma escala de abstração que incorpora como primeiro nivel os subsistemas. A imagem mostrada na figura, apresenta o primeiro nivel desta escala, no qual observam-se quatro grupos colaborativos ou subsistemas, denominados *MetaObjectManager*, *LuthierEventHandler*, *AbstactionLevel*, e *Component*. Estes grupos correspondem exatamente com os quatro sub-frameworks pelos quais Luthier é composto. Os nomes dos grupos são derivados da primeira classe que conforma o grupo. As setas indicam as colaborações existentes entre cada grupo, mantendo a codificação de cores descrita acima. Assim pode se observar, novamente, a seqüência habitual de execução estabelecida por Luthier.

Passando ao nivel seguinte na escala, aparecem as classes abstratas que compõem os grupos e aquelas classes que não formam parte de nenhum grupo identificado (FIGURA 7.22). Assim, o grupo *LuthierEventHandler* apresenta os relacionamentos entre os três tipos de componentes que definem o sub-framework, visões, manejadores de eventos e comandos, enquanto o grupo *MetaObjectManager* inclui *MetaObjectManager* e *MetaObjectClass*. Os grupos restantes estão constituídos só por uma classe abstrata.

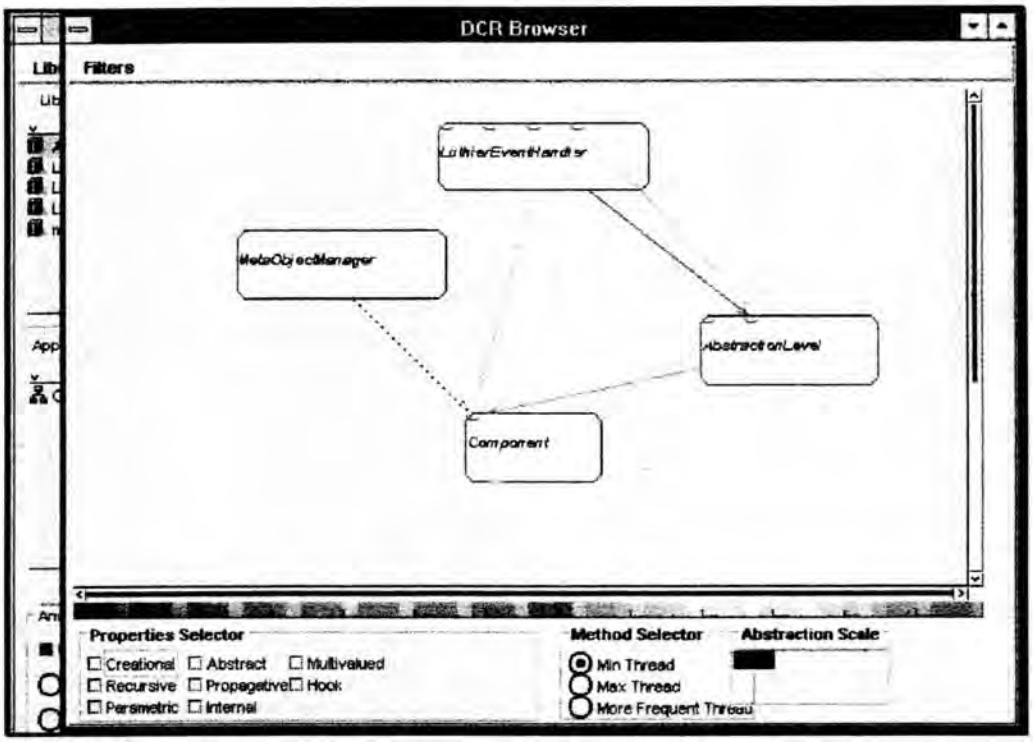


FIGURA 7.21- Visualização de subsistemas

Neste caso, pode observar-se que a classe *HypertextSystem* não forma parte de nenhum grupo, embora seja parte integral do sub-framework de hiperdocumentos. Isto é devido a que esta classe é uma servidora de funções para classes dentro de dois grupos diferentes (isto também indica que na programação das ferramentas, os componentes são acessados diretamente e não através da interface provida por esta classe). Já, as classes *LuthierBrowser* e *PropertiesSelector*, não formam efetivamente parte de nenhum grupo, pois foram desenvolvidas para cumprir papeis específicos na construção destas ferramentas, e não projetadas como parte do framework

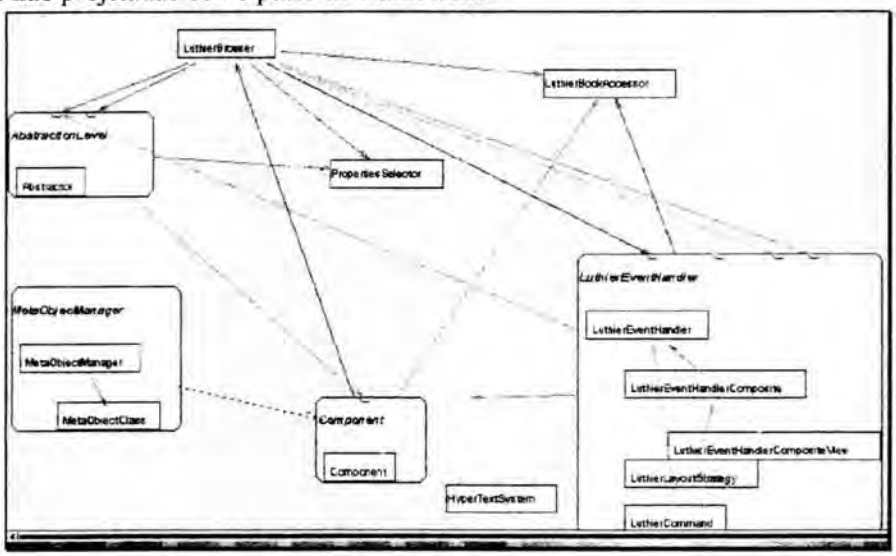


FIGURA 7.22- Visualização de subsistemas e classes abstratas componentes

A FIGURA 7.23 apresenta uma visão parcial do nível de hierarquia concreta, no qual aparecem as subclasses e seus relacionamentos. Para analisar detalhadamente os relacionamentos entre

os grupos, é possivel também, neste caso, utilizar os mecanismos de filtragem e zoom descritos na seção anterior.

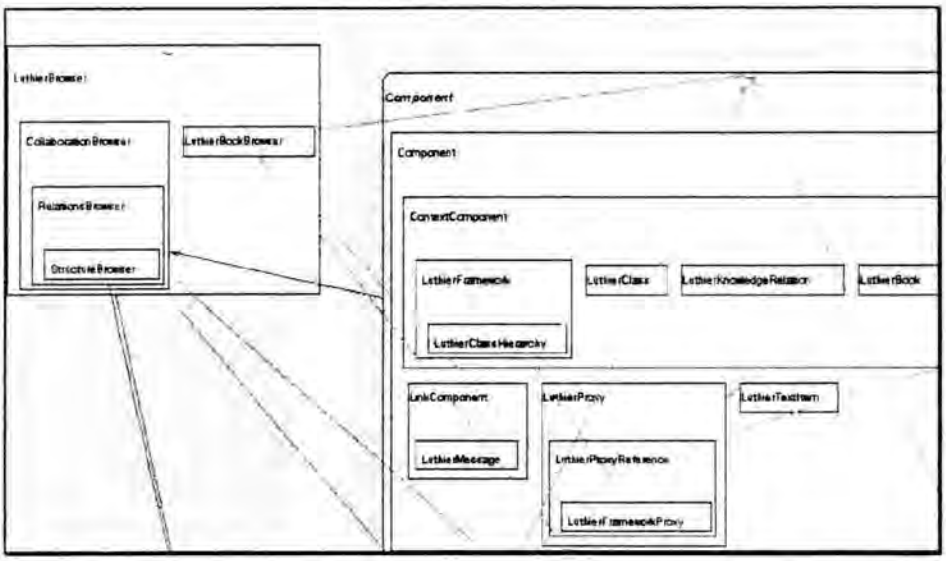


FIGURA 7.23- Visualização parcial da estrutura de um subsistema e hierarquias de classes

7.5.2 Análise de padrões de projeto

Luthier foi desenvolvido segundo o método habitual de desenvolvimento de um framework a partir da generalização de exemplos [JOH 93], mas com o objetivo de manter um alto grau de flexibilidade em tempo de execução. Assim, vários padrões de projeto foram aplicados para produzir uma estrutura inicial mais facilmente adaptável às mudanças.

Tomando em consideração as restrições discutidas no capitulo 4, Luthier oferece um suporte limitado para o reconhecimento de padrões de projeto, no sentido da utilização das carateristicas específicas de sua implementação em Smalltalk, o qual facilita seu reconhecimento, conforme é explicado mais adiante.

A FIGURA 7.24 apresenta uma imagem do browser de padrões provido pela ferramenta. Neste browser, utiliza-se da visualização OMT mostrada acima, substituindo a visualização dos relacionamentos pelo fluxo das mensagens trocadas entre os métodos que correspondem com cada padrão. O painel inferior apresenta a lista completa de padrões, ressaltando com cores diferentes os padrões que foram reconhecidos durante a análise. A seleção de um padrão nesta lista, produz que as classes envolvidas mostradas na visualização sejam coloridas com a cor que representa o padrão. Isto permite analisar cada padrão separadamente, e realizar o zoom para a visualização de fluxo de mensagens para analisar o comportamento dinâmico do padrão.

Alternativamente pode ser possível visualizar com cores os métodos e os relacionamentos que definem cada padrão. A primeira visualização é útil para centrar a atenção em padrões particulares, enquanto a segunda é útil para visualizar quais os padrões envolvidos no projeto de cada classe.

O exemplo da FIGURA 7.24 apresenta a visualização do padrão *FactoryMethod* na classe *LuthierLayoutStrategy*. Este padrão é sugerido porque o método abstrato *insert:at:model:* é redefinido nas subclasses *CollaborationLayoutStrategy*, *ComponentLayoutStrategy*, as quais criam, objetos do tipo *ComponentView*, pertencentes a uma hierarquia de classes diferente.

Já, a FIGURA 7.25, mostra a visualização alternativa dos padrões nos quais está envolvida a classe *LuthierLayoutStrategy*. Esta classe participa tanto do *FactoryMethod* como do *Strategy*. A classe *LuthierLayoutStrategy* é reconhecida com parte do Strategy porque encapsula o

algoritmo *relayoutGraphFor:*. As visões, *ComponentView* neste caso, referenciam alguma estratégia específica, *ComponentLayoutStrategy* no exemplo. e invocam este algoritmo para variar a distribuição espacial de seus componentes.

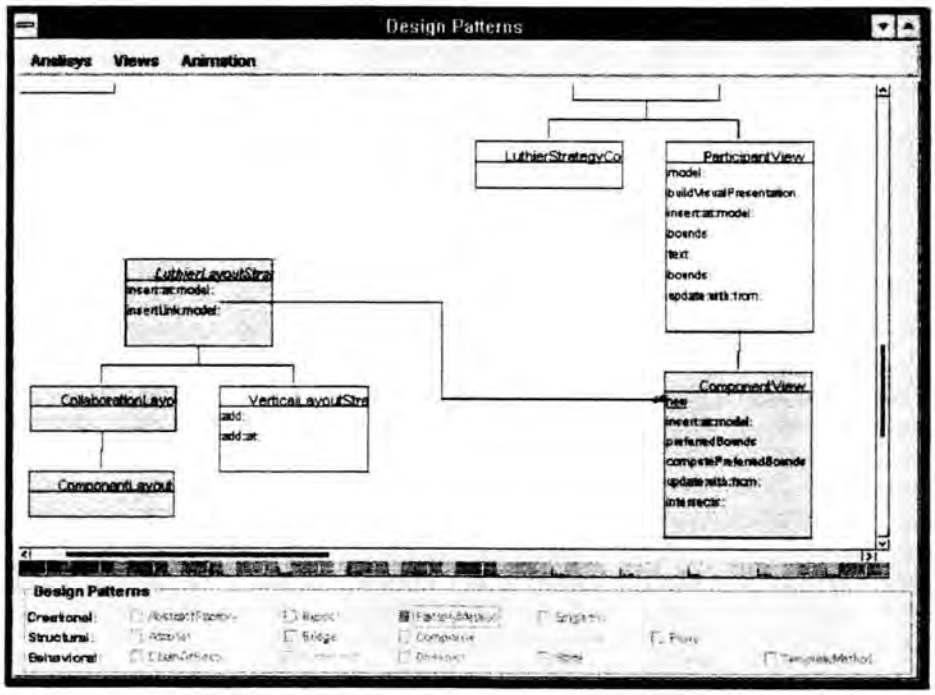


FIGURA 7.24- Visualização de padrões de projeto

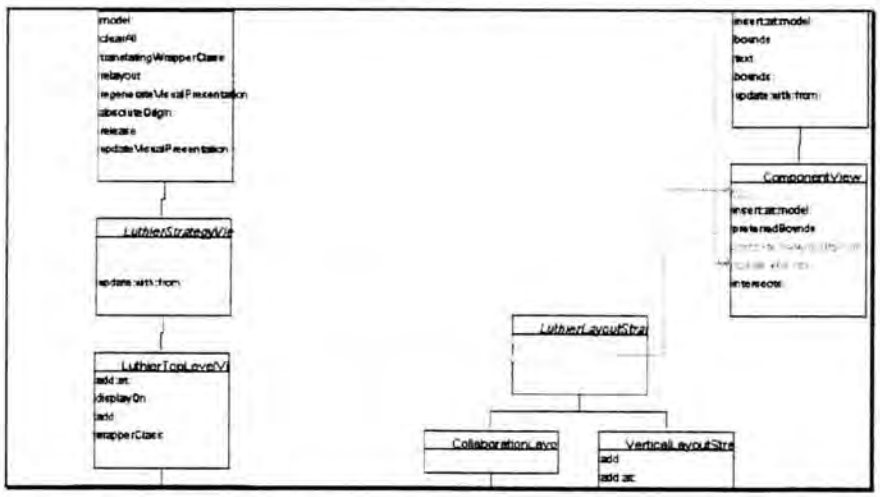


FIGURA 7.25- Visualização dos padrões *Strategy* e *Factory Method* na classe *LuthierLayoutStrategy*

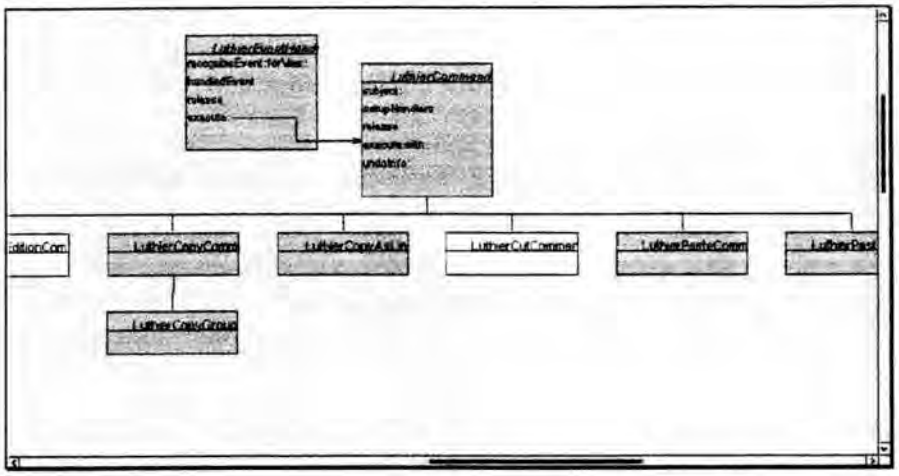


FIGURA 7.26- Visualização do padrão *Command* na classe LuthierCommand

A classe *LuthierCommand* apresenta um outro exemplo da interseção de vários padrões na mesma classe. Em primeiro lugar é reconhecido o padrão *Command* (FIGURA 7.26) porque as subclasses de *LuthierCopyGroupCommand*, *LuthierCopyAsLinkCommand*, *LuthierPasteText-Command*, *LuthierPasteCommand* e *LuthierCopyCommand* redefinem o método abstrato *execute:with:* para executar as ações correspondentes. Em segundo lugar, também pertence ao padrão *Composite*, pois a subclasse *LuthierEditionCommand* propaga mensagens para as subclasses anteriores, as quais são seus componentes. Esta classe coordena a atividade do comandos de edição, fornecendo por exemplo, o mecanismo geral de *undo* e de *buffer* de colado.

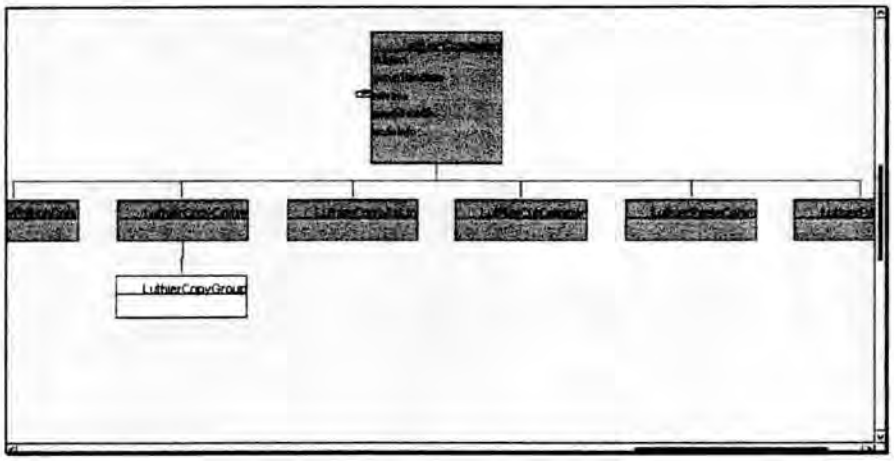


FIGURA 7.27- Visualização do padrão *Composite* na classe LuthierCommand

A FIGURA 7.28 apresenta a visualização, parcial, dos padrões envolvidos na classe *Abstractor*. A cor azul céu denota um padrão *ChainOfResponsisbility* existente na hierarquia de abstratores quando, se propagam através da hierarquia as mensagens *enable*, *topComponent* e *release*. Esta última mensagem é propagada desde as visões para as subvisões e também para os abstratores, razão pela qual, também é marcada como uma cadeia de responsabilidades. O método *getMessages* participa tanto de um padrão *Composite*, definido pela cor vermelha do relacionamento proveniente da classe *FrameworkAbstrator*. Esta classe, efetivamente, está composta por diferentes abstratores de mensagens para os quais propaga mensagens. Este método também forma parte do padrão *Observer*, identificado pela cor azul, pois anuncia mudanças que são tratadas pelo método *update:with:from:* definido na classe *Abstractor*. Efetivamente, os abstratores mantém dependências com os seus sujeitos

(os quais podem ser abstratores) para serem informados das mudanças produzidas. Finalmente o método *doesNotUnderstand:* é indicado como pertencente ao padrão Decorator, identificado pela cor de rosa, pois propaga mensagens ao seu sujeito, que pode ser um abstrator (seta recursiva) ou um *Component* (setas que saem da imagem).

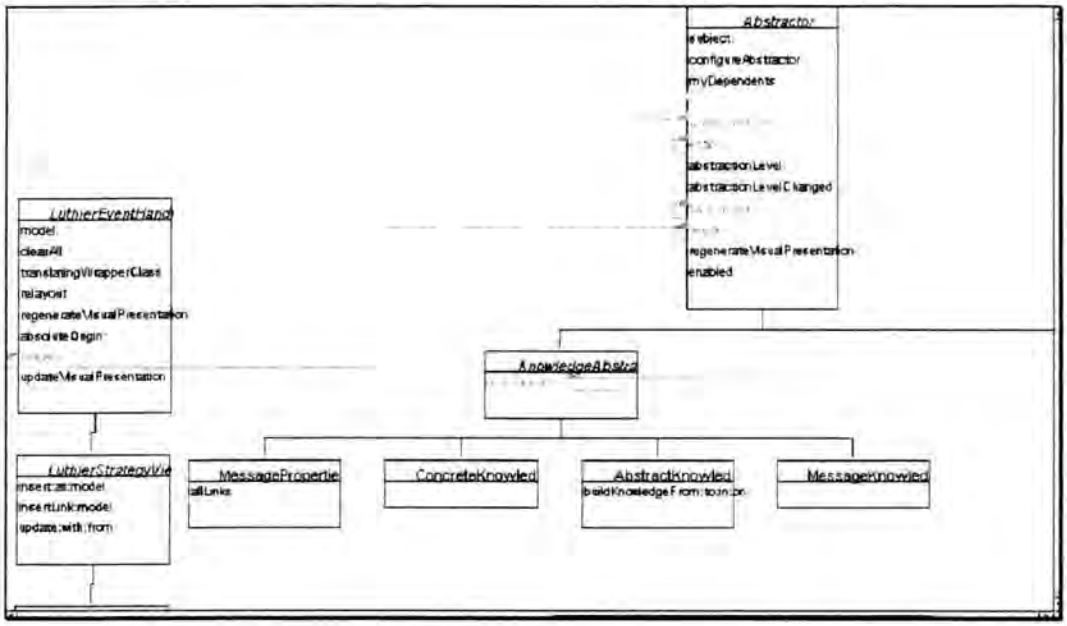


FIGURA 7.28- Visualização dos padrões da classe *Abstractor*

7.5.2.1 Aspectos de implementação

O reconhecimento dos potenciais padrões que podem existir na estrutura do framework analisado, é essencialmente um problema de *pattern matching*. Para isto, uma representação em termos de Prolog, é conveniente para facilitar a implementação destes algoritmos. Deste modo, segundo uma estratégia semelhante com a do *Program Explorer* [LAN 95], para a implementação do reconhecimento dos padrões utiliza-se a extensão de Smalltalk, OWB [AMA 96], a qual permite embeber código Prolog dentro de métodos Smalltalk normais.

A estratégia adotada para o reconhecimento dos padrões tira vantagens das caraterísticas próprias de Smalltalk seguintes:

- Os padrões *Decorator* e *Proxy* reconhecem-se se são implementados usando o método *doesNotUnderstand:*, a qual é forma mais comum de delegar responsabilidades entre objetos. Se um objeto recebe uma mensagem a qual não implementa, o método *doesNotUnderstand:* é automaticamente chamado pelo sistema de execução. Os padrões *Decorator* e *Proxy* habitualmente implementam-se redefinindo este método para passar a requisição ao seu componente.
- O *Observer* possui um objeto *Subject* que atualiza todos seus objetos dependentes, chamados de *observers*, cada vez que seu estado interno muda. Este mecanismo já está implementado em Smalltalk: na classe *Object*, a qual fornece o comportamento necessário para gerenciar a lista de dependentes de um objeto e informá-los das mudanças do objeto observado. Quando este objeto muda de estado anuncia a mudança enviando-se a mensagem *changed...*, a qual provoca que os objetos dependentes recebam a mensagem *update...*.

- Na representação da informação coletada, os métodos são classificados de acordo com sua categoria. Assim, o reconhecimento do padrão *TemplateMethod* é trivial.

O mecanismo geral de reconhecimento é encapsulado por um abstrator, o qual fornece a informação dos padrões à visualização. A representação Prolog da informação do modelo de hipertexto é convertida a formato Prolog, segundo a convenção seguinte:

```
class (ClassName, Superclass, RootClass )

message (OriginMethod, OriginClass,OriginRootClass, OriginMethodCategory,
        OriginMethodType, TargetMethod, TargetClass,TargetRootClass,
        TargetMethodCategory, TargetMethodType)
```

termos que representam a informação de classes e mensagens, necessária para realizar o reconhecimento.

As caraterísticas de cada padrão são codificadas por regras Prolog, que percorrem a base de dados na busca de combinações de mensagens e classes que satisfaçam a regra[5]:

```
Composite():-

    % Checks the existence of at least two subclasses to which messages are
    %propagted from a superclass

    message(Operation, Composite, FatherComposite, _, _,
                Operation, Component1, FatherComponet1, _, _, ),
    message(Operation, Composite, FatherComposite, _, _,
                Operation, Component2, FatherComponet2, _, _, ),
    hasCommonSuperClass( Composite, Component1 ),
    hasCommonSuperClass( Composite, Component2 ),
    assert( composite( Composite, Operation,
    [ [Component1, Operation], [Component2, Operation] ] ) ), fail.

Composite():- !.
```

Através destas regras é possível reconhecer casos relativamente evidentes da existência de um padrão, embora possam ser estendidas e aperfeiçoadas para reconhecer implementações não tão evidentes, acrescentando informação estática, como por exemplo, a estrutura de código dos métodos. Com esta informação poderia tratar-se de determinar com maior exatidão a existência de padrões não reconhecidos com as regras atuais.

7.6 Análise de Instâncias

As ferramentas descritas acima fornecem um repertório de capacidades adequado para realizar a análise da estrutura de um framework em diferentes níveis de abstração. Através de sua utilização combinada um usuário pode, gradualmente, alcançar a compreensão dos aspectos essenciais do framework, mas, podem ainda existir situações nas quais seja necessário realizar uma análise a nível de comportamento de instâncias para compreender como partes do framework são instanciadas.

Para esta tarefa, a visualização de grafo de fluxo de mensagens é de utilidade para instalar dinamicamente meta-objetos que implementem funções de análise sobre instâncias. Por exemplo:

- ***Análise de relacionamentos de instâncias***: Utilizando as capacidades de filtragem e de navegação sobre a estrutura de controle, é possível instalar meta-objetos que

[5] A implementação destas regras, e das porções específicas da interface, foi realizada por duas alunas da Universidade de Tandil, Argentina, como parte de seu trabalho de graduação sob minha orientação.

registrem aquelas instâncias envolvidas em uma dada seqüência de controle, as quais podem ser posteriormente analisadas através de um browser visual.

- ***Registro de instâncias***: Meta-Objetos que registrem todas as instâncias criadas durante a execução podem ser associados com um conjunto determinado de classes. Estas instâncias podem ser inspecionadas em diferentes pontos da execução, permitindo a análise incremental da sua evolução.
- ***Breakpoints sobre métodos***: Meta-objetos que implementam pontos de corte podem ser associados com métodos específicos. Se um método interceptado por um destes meta-objetos é ativado, informa à interface acerca dos objetos que enviam e recebem a mensagem, bem como seus argumentos e o valor de retorno. Desta forma é possível analisar a passagem de argumentos entre objetos e também analisar os resultados que a execução do método retorna.

A FIGURA 7.29 ilustra a ativação do retorno de um *breakpoint* estabelecido sobre o método *on:* da classe *LuthierProxy*, para analisar o comportamento de criação de *proxies*, durante a criação de um livro (ocultado parcialmente pela janela do browser). No exemplo, a interface mostra textualmente uma instância de *LuthierBookChapter* como a originadora da mensagem, o estado de suas variáveis de instância, a instância de *LuthierClassHierarchy* passada como argumento, a classe *LuthierProxy* receptora da mensagem e a instância dessa classe retornada pelo método. Neste ponto o usuário pode inspecionar o estado de quaisquer destes objetos, além de utilizar as funcionalidades da visualização já descritas, ou instalar novos pontos de corte.

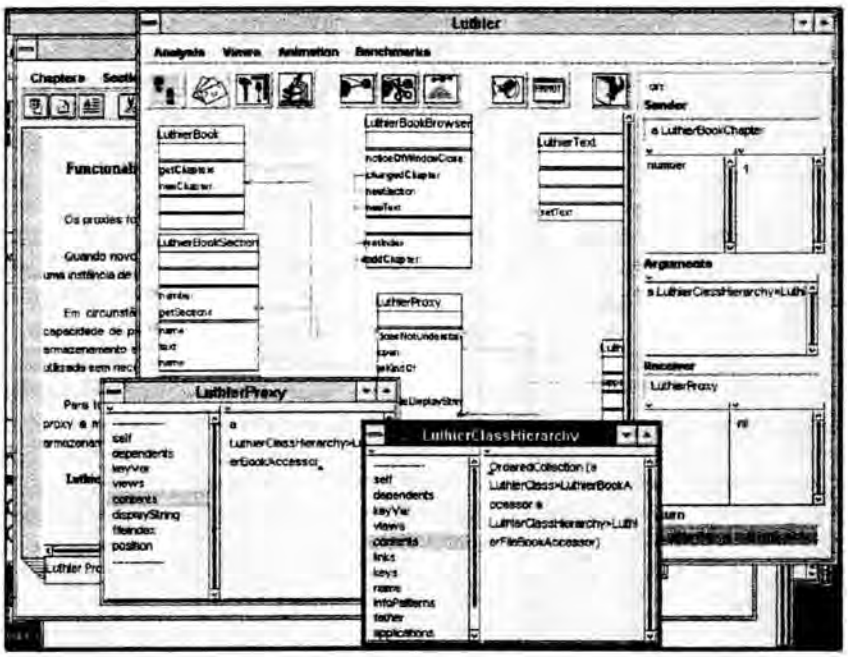


FIGURA 7.29- Exemplo de ativação de um *breakpoint*

FIGURA 7.30 mostra o browser de instâncias inspecionando as instâncias de *LuthierBook* criadas até esse ponto da execução do exemplo. Estas instâncias são coletadas por um meta-objeto associado com a classe *LuthierBook*, para registrar todas as instâncias criadas pela classe. O browser mostra uma visão gráfica das instâncias, composta por suas varáveis de instância agrupadas de acordo com a classe na qual são definidas dentro da hierarquia. Uma seta desde uma variável para uma instância indica que uma referência a essa instância e mantida nessa variável. Com esta

visualização pode explorar facilmente, selecionando variáveis com o mouse, como é organizado o conteúdo de um livro, em termos de capítulos que contém um item de texto com o título e seções que contém itens de texto e subseções. Cada um destes elementos é representado por um *proxy*, tal como foi deduzido no exemplo da animação arquitetônica da seção 7.2.2.1. Desta forma, através da análise da organização de instâncias o usuário pode confirmar a existência de estruturas sugeridas pelas outras ferramentas.

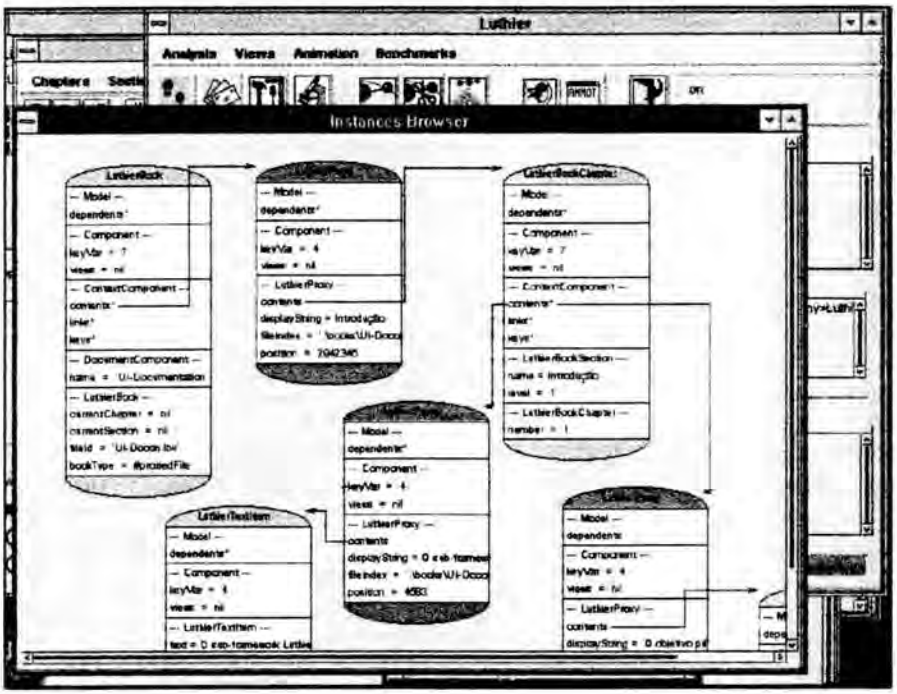


FIGURA 7.30- Browser de instâncias

7.7 Construção de Livros de Documentação

O suporte para a construção de livros de documentação, construído em base dos sub-frameworks de hiperdocumentos e de visualização, fornece capacidades limitadas para edição de texto, colado de texto externo, formatação e destaque de texto de acordo com estilos predefinidos e inserção de diagramas copiados ou vinculados das outras visualizações.

As capacidades de formatado e destaque de texto com diferentes fontes e cores, permite organizar uma apresentação que destaca visualmente informação relevante ou vínculos para outras páginas ou representações, os quais são naturalmente suportados pela organização de hiperdocumento.

Através das funções de colado, é possível selecionar partes de uma visualização e inseri-las em uma página, de acordo com o tipo de visualização que defina o *template* utilizado para representá-la. Um *template* é um tipo de visão parametrizada que recebe informação a ser visualizada e constrói uma visualização utilizando uma dada notação. Por exemplo, é possível copiar uma classe do grafo de fluxo de mensagens e inseri-la utilizando um *template* que gera uma representação estática OMT, ou um *template* que gera uma visualização do código dessa classe.

A FIGURA 7.31 apresenta uma imagem da edição do livro de documentação do framework de interfaces. O livro está constituído, até o momento por três capítulos. O capítulo corrente explica a funcionalidade de manipulação direta, através de um texto introdutório e duas seções, *EventHandlers* e *Comandos*. O texto introdutório foi colado diretamente da explicação desta função dada no capítulo anterior.

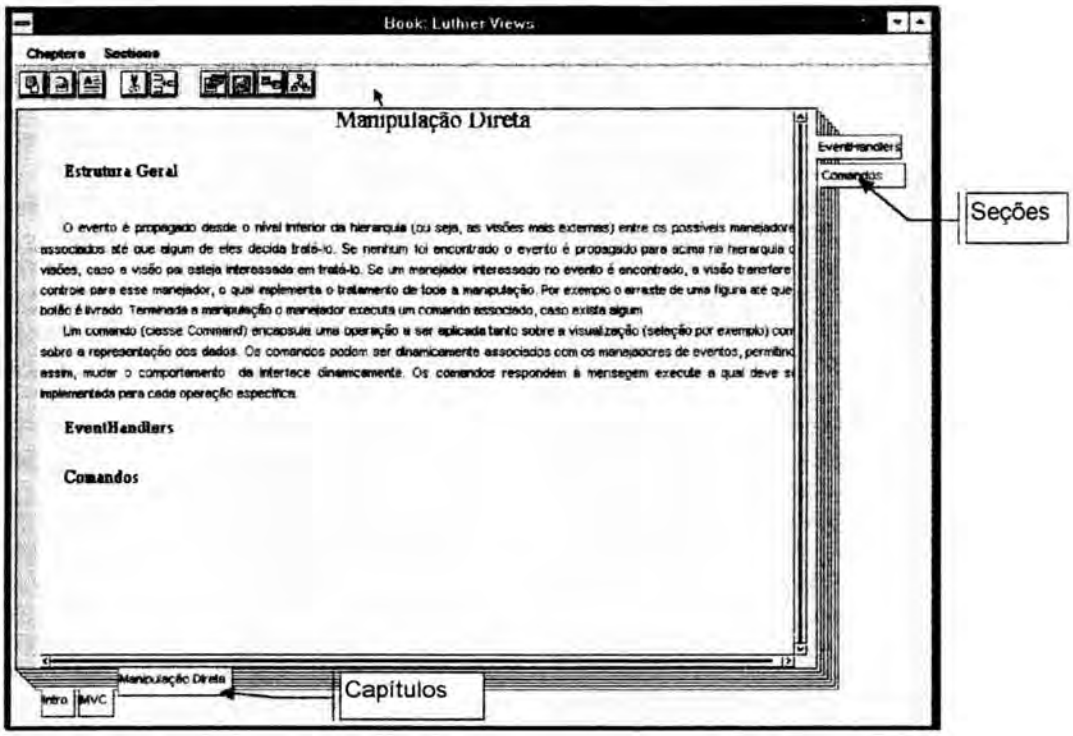


FIGURA 7.31- Exemplo de página de um livro de projeto

A FIGURA 7.32 mostra a imagem da explicação da estrutura dos manejadores de eventos, na qual foi inserido um *template* de OMT, para explicar a estrutura de classes da classe *EventHandler*. Já a FIGURA 7.33 mostra o processo de copia de um grupo de métodos da representação no grafo de fluxo de mensagens para serem colados na página utilizando um template que recupera o código desses métodos. Também observa-se a definição da classe que fora inserida antes.

FIGURA 7.32- Inserção de estrutura de classes de um framework utilizando OMT

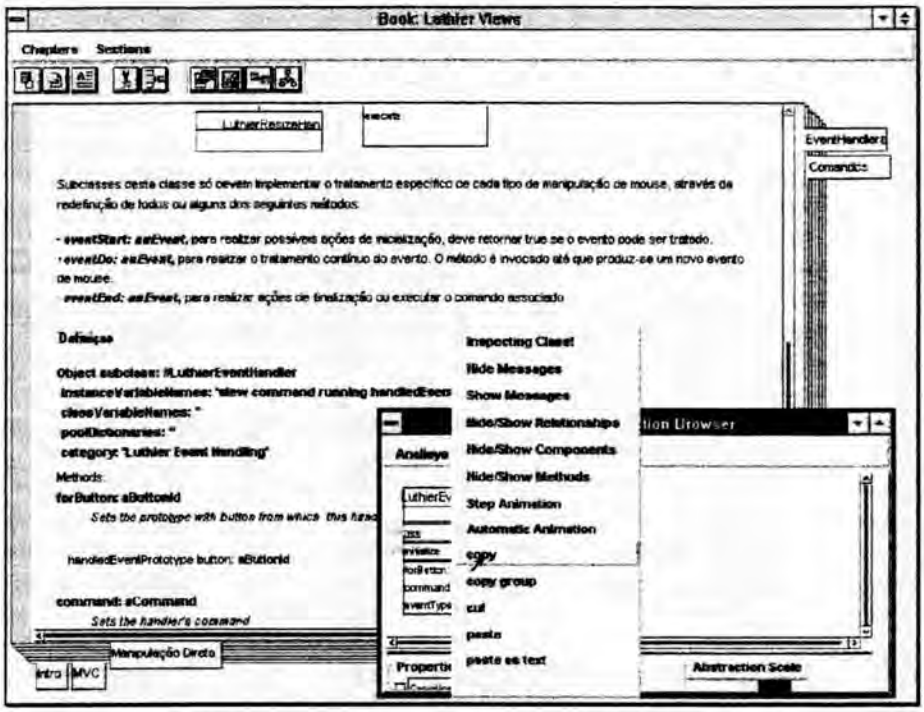


FIGURA 7.33- Exemplo de inserção de código

Uma aplicação importante do suporte para a construção de livros é a geração automática de explicações acerca da informação reconhecida por ferramentas, como é o caso dos padrões de projeto. O browser de padrões permite a geração de um livro que contém os padrões reconhecidos no framework analisado, com uma explicação sintética do objetivo do padrão, e para cada instância de padrão reconhecida, porque as classes é métodos envolvidos são parte desse padrão, bem como o diagrama OMT que descreve a localização do padrão na estrutura de classes.

O livro é dividido em três capítulos, um para cada categoria de padrão, isto é criacionais, estruturais e comportamentais. Cada capítulo é dividido em seções, as quais contém a explicação correspondente a cada padrão reconhecido.

A FIGURA 7.34 mostra um exemplo da página correspondente com a seção do padrão *Chain of Responsibility*, na qual aparecem o objetivo do padrão e a lista de instâncias reconhecidas. Na figura observa-se a explicação para a cadeia existente na classe *Abstractor*, dada pela propagação do método *topComponent*, entre as classes *HierarchyAbstractor* e *FrameworkAbstractor*, e a cadeia da classe *Component* dada pela propagação da mensagem *dumpContents* entre as classes *LuthierBook* e *LuthierProxy*.

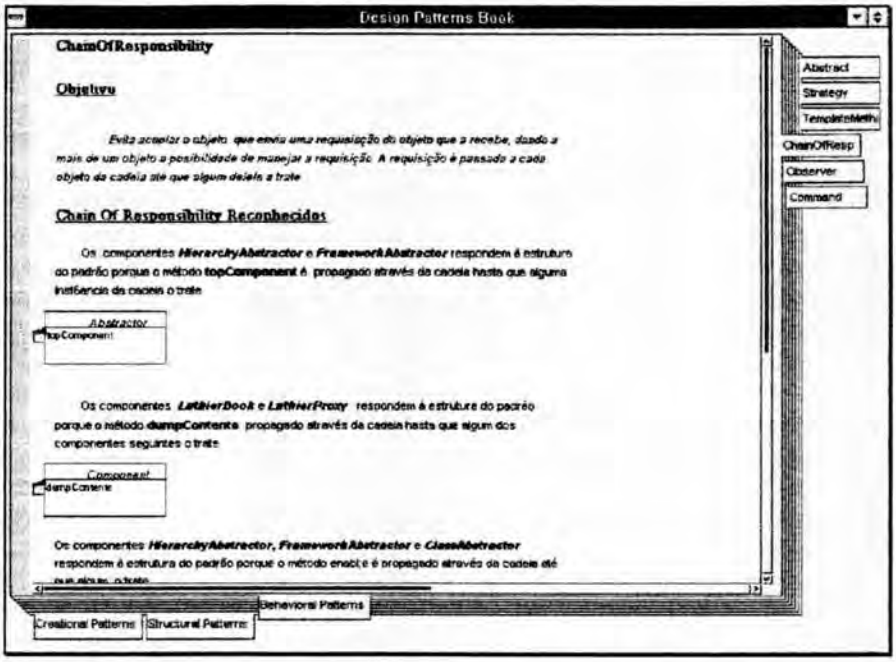


FIGURA 7.34- Página com a explicação dos padrões *Chain of Responsibility* reconhecidos

8 Experimentos

No capítulo precedente apresentou-se uma ferramenta projetada para apoio à compreensão de frameworks, desenvolvida com o suporte provido por Luthier. O desenvolvimento desta ferramenta, mostra as capacidades do framework para construir ferramentas de análise de programas. No entanto, a eficácia destas ferramentas para apoiar na compreensão de frameworks, só pode ser comprovada empiricamente, através de sua utilização em casos reais. Por esta razão, foram realizados dois experimentos na utilização de dois frameworks para a construção de aplicações. A abrangência destes experimentos não permite demonstrar com precisão o grau de eficácia da abordagem de compreensão de frameworks nem das ferramentas construídas para suportá-la, mas serve para obter uma idéia aproximada tanto das suas vantagens como de suas limitações.

Os experimentos foram realizados na Universidade de Tandil, Argentina, com um grupo seleto de alunos do terceiro ano da carreira de Engenharia de Sistemas, como parte das atividades práticas da disciplina de Programação Orientada a Objetos. Os alunos foram selecionados entre os que mostraram na disciplina o melhor desempenho na resolução de problemas e em programação Smalltalk[1].

A medição de resultados de um dos experimentos foi realizada utilizando uma ferramenta para a coleta de métricas de código, desenvolvida utilizando Luthier por outro grupo de alunos. As diferenças de projeto entre as diferentes aplicações desenvolvidas foi analisada utilizando a própria ferramenta de análise de frameworks.

Os experimentos foram planejados para permitir obter dados acerca da eficácia da ferramenta bem como do impacto de aprendizado da mesma. No primeiro experimento participaram dois grupos, um utilizando a ferramenta e o outro não. Já no segundo experimento acrescentou-se mais um grupo que não utilizou a ferramenta, passando os outros dois a utilizá-la.

As seções seguintes, apresentam uma descrição sintética dos experimentos realizados e a análise de resultados obtidos para cada uma delas. Finalmente descreve-se, sucintamente, a ferramenta de métricas desenvolvida.

8.1 Experimento 1: Uso de um Framework para Jogos de Tabuleiro

Esta primeira experiência teve como objetivo principal introduzir um grupo de alunos na utilização de Luthier para a produção de uma aplicação utilizando um framework muito simples para a construção de jogos de tabuleiro. O framework é constituído só por uma classe abstrata, que representam o jogo e três classes concretas para as peças, o tabuleiro e o controle de mouse (FIGURA 8.1).

O experimento consistiu no desenvolvimento de uma interface para um jogo de xadrez que forneça capacidades para movimentar as peças com manipulação direta, tomando em consideração posições válidas para cada peça, e *feedback* visual para as posições válidas que uma dada peça pode ocupar quando está sendo deslocada.

O tempo estabelecido para o desenvolvimento foi de três semanas, com um grupo (G1) utilizando uma versão de Luthier, muito mais reduzida que a atual, e um outro grupo (G2) utilizando só a documentação disponível do framework. O grupo G1 não teve conhecimento prévio da ferramenta antes de começar a experiência. Deste modo, tentou-se ter uma idéia aproximada do impacto do aprendizado do uso da ferramenta, em conjunto com a sua utilização para compreender um framework.

---

[1] Os experimentos foram monitorados com a colaboração do Ms.C. Álvaro Ortigosa, atual professor da disciplina.

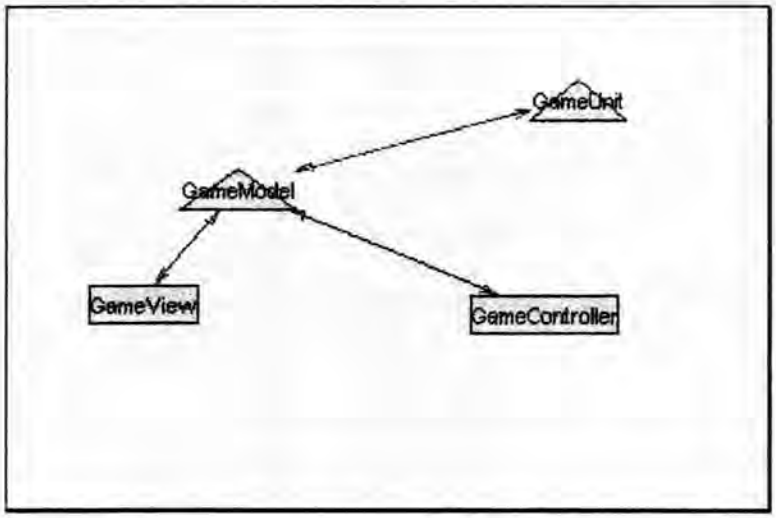


FIGURA 8.1- Framework de Jogos

As aplicações produzidas por ambos grupos foram praticamente semelhantes no que diz respeito à estrutura e subclasses criadas, mas na média, a implementação do G2 foi aproximadamente 30% maior em quantidade de linhas de código. Isto foi devido a três fatos relevantes:

- A distribuição da funcionalidade entre as classes que implementam a manipulação direta foi totalmente diferente. Enquanto o grupo G1 utilizou um cópia adaptada de um controlador de tabuleiro de um exemplo, o grupo G2 desenvolveu um controlador que concentrava toda a funcionalidade de manipulação, deste modo, não reutilizaram os serviços de manipulação providos por outras classes do framework.
- A não redefinição de um método abstrato para informar o tamanho das peças no tabuleiro, conduziu ao grupo G2 à criação de uma estrutura de controle própria para deslocamento das peças.
- Muitas porções de implementação das visualizações foram realizadas pelo G2 utilizando recursos próprios de Smalltalk, e não os fornecidos pelo framework, os quais contemplavam, por exemplo, deslocamento de peças dentro de um tabuleiro.

Apesar da natureza informal do experimento, ele permite extrair como conclusão relevante que o grupo G1 produziu uma aplicação com maior nível de reutilização de funcionalidade e que respeitava totalmente a estrutura dos exemplos providos com o framework, utilizando praticamente o mesmo tempo de desenvolvimento que o G2. Levando em conta que este grupo não conhecia a ferramenta, pode-se concluir que desenvolveram a aplicação em um tempo relativo menor que o grupo G2.

Este experimento resultou importante para definir as limitações que a ferramenta apresentava. Essencialmente, o mecanismo de análise de instâncias foi desenvolvido pela tendência que foi detectada de pensar o comportamento dos exemplos em termos de instâncias.

8.2 Experimento 2: Uso de um Framework para Editores Gráficos

O segundo experimento foi planejado para realizar uma monitoração de tempos utilizados, agora por três grupos, os dois grupos anteriores, GL1 e GL2 usando a ferramenta, mais um terceiro (GN) que não a utilizaria.

O experimento consistiu na construção de um editor de Redes de Petri, utilizando um framework muito mais complexo, *HotDraw*, para construção de editores gráficos. O experimento foi realizado sob controle estrito, tendo os grupos assumido o compromisso de não trocarem informação acerca de decisões de projeto entre si.

8.2.1 HotDraw

HotDraw [JOH 92] é um framework para a criação de editores gráficos para figuras bidimensionais, podendo ser utilizado para construir editores para gráficos especializados, tais como esquemas de hardware, ou diagramas de projeto de software. As figuras se desenham em uma tela de edição (*drawingview*) selecionando uma das ferramentas da paleta de ferramentas. O conjunto inicial de ferramentas para o desenho de figuras inclui: linhas; setas; retângulos; círculos; e texto. As operações de manipulação destas figuras incluem: seleção; redimensionamento; deslocamento; remoção e agrupamento.

Os usuários de um editor construído com HotDraw podem manipular figuras de diversas maneiras, interagindo diretamente com elas através do mouse. A interação direta com uma figura é realizada selecionando a figura com a ferramenta de seleção e manipulando os pontos de controle (*Handles*). Estes pontos de controle permitem mudar o tamanho e a forma das figuras. Também é possível abrir um menu de operações que podem realizar-se sobre a figura, como por exemplo mudar sua cor. Outra caraterística de *HotDraw* é a habilidade de manter restrições entre as figuras. Por exemplo, um arco que liga duas figuras depende da posição delas e, portanto, quando uma é deslocada o arco será também deslocado para manter a restrição. As restrições, representadas pela classe *HotDrawConstraint*, são baseadas em um suporte de gerenciamento de sistemas de restrições provido por um outro framework, *SkyBlue*. O gerenciamento de restrições não precisa, no caso geral, ser redefinido, sendo utilizado diretamente através da criação de restrições associadas com figuras[2].

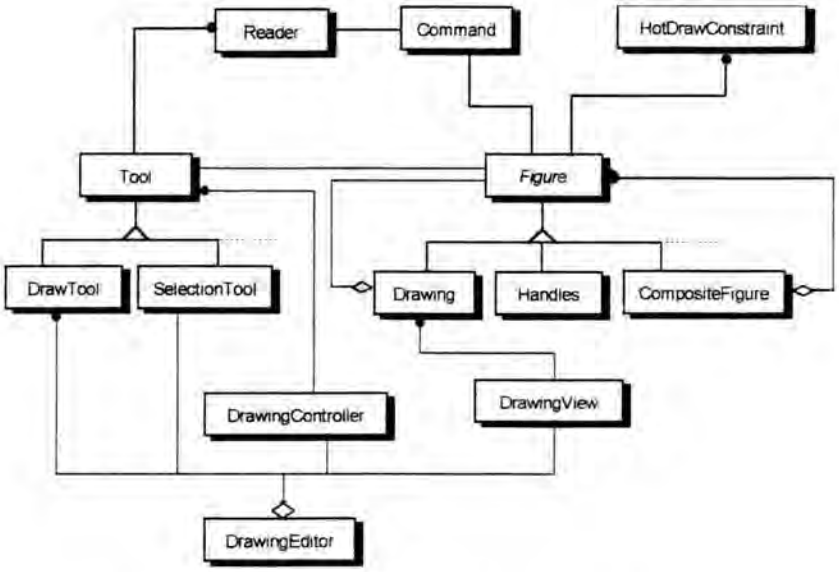


FIGURA 8.2- Estrutura de classes de *HotDraw*

A FIGURA 8.2 apresenta um diagrama resumido de classes de *HotDraw*. A classe *Figure*, única classe abstrata do framework, representa os objetos que se desenham com as ferramentas; a classe *Tool* representa os objetos que aparecem no menu de ferramentas e manipulam as figuras; a classe *Handles* representa os objetos que se utilizam como pontos de controle que aparecem sobre as figuras. As classes *Reader* e *Command* implementam mecanismos de tratamento de eventos e execução de comandos. A classe *DrawingController* representa os objetos que administram a interação do usuário na tela de edição ou área de desenho; esta última é representada pela classe *DrawingView*. Com a estrutura fornecida pelo framework é possível criar editores específicos, utilizando as classes

[2] Aos fins deste exemplo a descrição deste sistema de restrições é desnecessária e não pode ser resumida facilmente devido a sua complexidade. O leitor interessado pode encontrar uma descrição em [JOH 92].

concretas existentes (*SelectionTool* por exemplo) e criando novos componentes através da especialização de outros, caso não existam na biblioteca. Geralmente, cada nova aplicação será representada por uma classe encarregada de criar as instâncias dos componentes e ligá-los para iniciar a execução, representada por *DrawingEditor* no diagrama.

8.2.2 Requisitos da aplicação desenvolvida

A aplicação desenvolvida utilizando *HotDraw* consistiu de um editor para Redes de Petri com a funcionalidade seguinte:

- Criação de figuras: *Transições*, com forma retangular, um texto interno (nome) e um externo (condição) ao retângulo. *Estados*, com forma elíptica e dois textos internos. *Conexões*, uma seta que pode unir só um estado e uma transição e vice-versa; com uma texto associado representando os parâmetros.
- Manipulação direta de criação, deslocamento e redimensionamento de figuras.
- Comandos de edição básicos com volta atrás (*undo*), ou seja, inserção, apagado, etc.
- *Feedback* para a criação de conexões que indique os destinos válidos de uma dada conexão a partir de uma determinada origem.

A FIGURA 8.3 apresenta uma imagem de um editor construído por um dos grupos. Existiu um controle estrito sobre a funcionalidade implementada por cada grupo, de forma tal a se poder comparar resultados com maior precisão.

A definição das figuras a serem editadas e a utilização do sistema de restrições são os aspectos mais importantes do projeto destes editores, pois o resto da funcionalidade está praticamente implementada pelo framework, e é simples de ser copiada dos exemplos existentes. Deste modo, através do experimento foi possível testar a compreensão do framework tanto nos aspectos de redefinição de classes quanto da utilização de classes prontas.

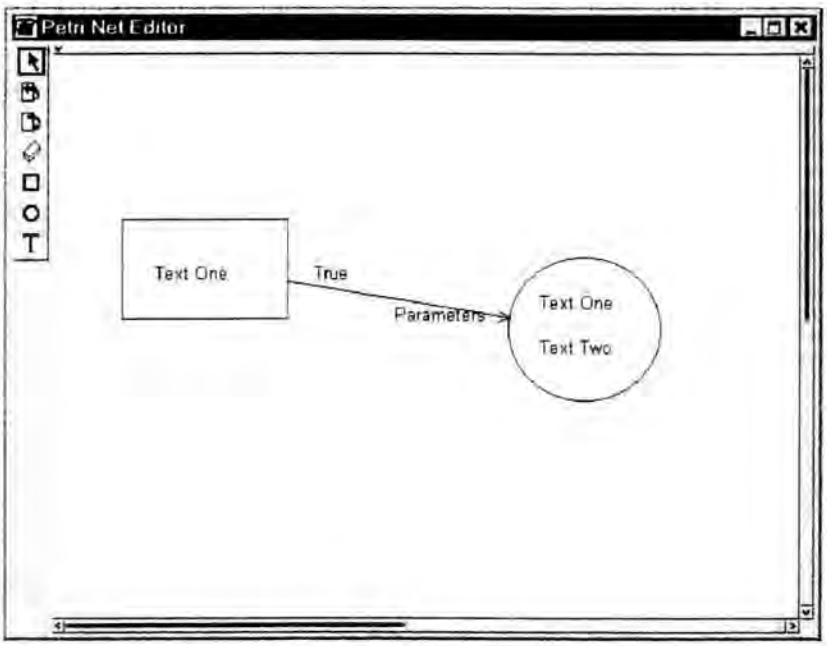


FIGURA 8.3- Editor de Redes de Petri construído por um dos grupos

8.2.3 Medição de resultados

Para comparar os resultados de cada grupo utilizou-se uma ferramenta desenvolvida com Luthier para a toma de métricas de código. Esta ferramenta implementa as métricas descritas em

[LOR 94], das quais, só um subconjunto relevante para analisar diferenças entre as aplicações desenvolvidas com um framework foi tomado em consideração (TABELA 8.1).

Os resultados destas medições admitem muitas interpretações diferentes e são altamente dependentes do tipo do programa analisado. Particularmente, para o caso da comparação do grau de reutilização obtido de um framework, a maior parte delas não aporta informação que possa ser considerada como muito significativa. No entanto, oferecem sugestões interessantes das diferenças relativas entre as três aplicações desenvolvidas, as quais podem ser verificadas através de outras análises. Nas métricas descritas na tabela, são ressaltadas aquelas que, na media por classe, são particularmente interessantes como sugestão do grau de reutilização de um framework obtido por diferentes aplicações, bem como qualidade de projeto relativa entre essas aplicações.

TABELA 8.1- Métricas consideradas para a avaliação

Métrica	Significado
Número de Classes	Número de classes da aplicação. Neste caso só foram consideradas as classes agregadas por cada aplicação e não as do framework.
Linhas de Código	Quantidade total de linhas do programa
Número de Instâncias	Numero de instâncias criadas durante a execução dos exemplos.
Nível de Aninhamento	Máximo nível de aninhamento encontrado em todas as classes analisadas, ou seja, a quantidade de superclasses até Object.
Número de Métodos	Número total de métodos
Número de Métodos Redefinidos	Soma dos métodos redefinidos por todas as classes analisadas.
Número de Métodos Agregados	Soma dos métodos que não são redefinidos, isto é, que não são definidos nas superclasses.
Número de Métodos Herdados	Para cada classe, somam-se todos os métodos definidos por suas superclasses.
Taxa de Especialização	Soma da relação entre número de métodos redefinidos multiplicado pelo nível de aninhamento da classe dividido pelo número de métodos total próprio da classe (redefinidos + agregados). A idéia desta métrica é que o número de métodos redefinidos não deve ser grande, principalmente para classes com um nível de aninhamento alto. Neste caso é aconselhável que este número seja baixo, mas no caso dos frameworks isto é relativo, pois podem existir muitos métodos a serem redefinidos.
Número de Sentenças	Quantidade de sentenças no código. No caso de Smalltalk são aquelas terminadas com ponto.
Número de Mensagens	Soma da quantidade de mensagens de cada método. Em Smalltalk uma sentencia pode conter vários mensagens. Assim esta métrica é mais relevante para dar idéia de complexidade de código.
Número de Parâmetros	Soma dos argumentos que recebe cada método.
Número de Variáveis de Classe	Soma das variáveis de classe definidas em todas as classes
Número de Variáveis de Instância	Soma das variáveis de instância definidas em todas as classes
Número de Classes sem Instâncias	Superclasses que não possuem instâncias na execução. Esta métrica depende da seqüência de execução, mas, no caso dos frameworks, dá uma idéia relativa da criação de hierarquias, as quais são no caso geral desnecessárias quando o framework é bem projetado.
Número de Mensagens Recebidas	Número total de mensagens recebidos por uma classe durante a execução.
Número de Métodos não Chamados	Número de métodos não chamados durante a execução. Esta métrica pode indicar métodos não chamados nunca, bem como funcionalidade específica da aplicação, a qual não segue a estrutura de controle definida pelo framework. Neste caso pode representar uma sugestão, muito relativa, de escassa reutilização do framework.

Métrica	Significado
Número de Métodos Públicos	Métodos chamados desde outras classes
Número de Métodos Privados	Métodos só chamados desde a própria classe que os define.
Número de Locais Variáveis	Variáveis locais aos métodos. É de utilidade para ter uma idéia relativa da complexidade dos métodos.

No caso geral, é considerado que um projeto adequado deveria acrescentar comportamento em subclasses, redefinindo uma baixa quantidade de métodos. Isto, não necessariamente é correto quando se trabalha com frameworks bem desenvolvidos, nos quais poucos métodos adicionais supõe-se serem necessários. Obviamente, esta consideração é muito relativa para cada framework em particular, e dependente se o framework é projetado para ser utilizado por herança ou composição.

No entanto, para comparar aplicações, uma relação entre o número de métodos redefinidos e os agregados é de utilidade para dar uma sugestão do grau de reutilização obtido, especialmente, se o framework é baseado em herança. Neste caso, a relação deveria tender a infinito para o caso de um framework *ideal*, no qual só seja necessário implementar métodos públicos abstratos ou *hook* (uma classe pode acrescentar método privados por razões de projeto interno, que não estendem o protocolo do framework). Por esta razão, nas métricas utilizadas definiu-se a métrica *Proporção Redefinidos/Agregados*, para representar esta relação. Um valor alto desta relação pode sugerir um maior grau de reutilização do framework[3].

A TABELA 8.2 apresenta os valores das métricas correspondentes aos três editores desenvolvidos. As métricas de tempos de execução correspondem às medições coletadas na seqüência seguinte:

- Criação de um estado
- Criação de uma transição
- Criação de uma conexão entre ambas figuras

Esta seqüência envolve praticamente 80% da funcionalidade específica requerida das aplicações.

Nos valores da tabela nota-se de forma evidente uma diferencia substancial no número de classes criadas pelos grupos que utilizaram a ferramenta e o grupo que não a utilizou. Isto representa uma forte sugestão de diferenças substanciais de projeto e portanto um grau de compreensão do framework muito inferior do grupo GN. Também podem-se perceber diferenças, não muito significativas, entre os grupos GL1 e GL2, particularmente na quantidade de linhas de código e quantidade de métodos agregados.

[3] A sugestão da implementação desta métrica foi do Prof. Álvaro Ortigosa

TABELA 8.2- Valores das métricas

Métrica	GL1	GL2	GN
Totais			
Número de Classes	**16**	**12**	**32**
Linhas de Código	1427	980	1854
Número de Instâncias	28	30	40
Nível de Aninhamento (max)	7	7	8
N° de Métodos	**178**	**148**	**281**
N° de Métodos Herdados	2040	1569	4082
N° de Métodos Redefinidos	102	95	126
N° de Métodos Agregados	**76**	**53**	**155**
N° de Sentencias	700	718	1217
N° de Mensagens	**815**	**781**	**1370**
N° de Parâmetros	106	83	158
N° de Variáveis de Classe	**0**	**0**	**16**
N° de Variáveis de Instância	23	12	15
N° de Classes sem Instancias	**9**	**4**	**18**
N° of de Mensagens Recebidas	22162	5745	14397
N° de Métodos não Chamados	**51**	**25**	**115**
N° de Métodos Públicos "	94	**75**	125
N° de Métodos Privados "	33	38	41
N° de Variáveis Locais	140	**89**	140
Medias			
N° Métodos Herdados/Classe	127.50	**130.75**	127.56
N° Métodos Redef./Classe	**6.37**	**7.91**	3.93
N° Métodos Agregados/Classe	8.56	**4.41**	6.85
Proporção R/A	0.63	**1.55**	0.61
Índice de Especialização	**3.47**	**3.20**	**2.36**
N° de Mensagens/Classe	50.93	65.08	42.81
N° de Parâmetros/Classe	**6.62**	**6.91**	4.93
Linhas de Código/Classe	89.18	81.67	**57.93**
Linhas de Código/Método	8.215	**6.62**	7.23
N° de Mensagens/Método	**4.58**	5.27	4.87

O índice de especialização de ambos grupos é praticamente equivalente, enquanto o índice do GN é bastante menor. Este índice indica um baixo grau de redefinição de métodos e de aproveitamento de funcionalidade do framework, pois o número de métodos agregados é grande. Na média, no entanto, pode perceber-se que o GL1 apresenta o maior número de métodos agregados por classe e o maior nível de aninhamento. Entretanto o índice de especialização é complementado por um alto número de métodos redefinidos. A relação redefinidos/agregados, por sua vez, mostra uma paridade entre o GL1 e o GN, enquanto o valor do GL2 é o dobro dos outros. Ambos índices combinados sugerem um melhor grau de reutilização do grupo GL2.

As métricas restantes não oferecem informação significativa, com a exceção dos aspectos seguintes:

- O alto número de variáveis de classe definidas pelo GN, o qual é absolutamente desnecessário
- O GL2 apresenta o menor número de classes sem instâncias e de métodos não chamados, sugestão de um melhor aproveitamento de funcionalidade,

Estes valores sugerem um melhor desempenho do grupo GL2, mas não permitem assegurar, de modo algum, que a qualidade da solução e de reutilização seja superior, principalmente entre GL1 e GL2. Para isto é necessário uma análise detalhada das diferenças de projeto entre os editores que as métricas não refletem. Esta análise é apresentada na seção seguinte ressaltando as diferenças significativas.

8.2.4 Comparação geral dos projetos

Tal como as métricas sugerem, da análise das aplicações utilizando Luthier, surge que a aplicação do grupo GN apresenta os maiores problemas relacionados com o projeto do editor, enquanto os grupos que utilizaram a ferramenta apresentam poucas diferenças entre si.

Comparados de forma geral os resultados do três grupos a diferencia essencial nas decisões de projeto é relativa ao projeto das figuras a serem editadas e a utilização das restrições.

- GL1 apresenta a melhor estrutura em termos de reutilização de funcionalidade do framework, devido à seleção adequada das classes das quais herdar comportamento,

TABELA 8.3- Classes definidas por GL1

	Classes	Superclasse
1	GL1ConnectionArc	SplineFigure
2	GL1EllipseFigure	EllipseFigure
3	GL1RectangleFigure	RectangleFigure
4	GL1ConnectionCommand	ConnectionCommand
5	GL1Drawing	Drawing
6	PetriNetEditor	DrawingEditor
7	GL1FigureCreationCommand	FigureCreationCommand
8	GL1InText	TextFigure
9	GL1GroupFigure	GroupFigure
10	GL1OutText	TextFigure
11	GL1TextFigureCommand	TextFigureCommand
12	GL1TextFigureSelectionCommand	TextFigureSelectionCommand
13	GL1CommandHandle	CommandHandle
14	GL1ConstraintDrawing	ConstraintDrawing
15	GL1TextTool	TextTool
16	GL1DrawingCommand	DrawingCommand

TABELA 8.4- Classes definidas por GL2

	Classe	Superclasse
1	GL2Connection	CompositeFigure
2	GL2State	CompositeFigure
3	GL2Transition	CompositeFigure
4	GL2PetriConnectionCommand	ConnectionCommand
5	GL2PetriDrawing	Drawing
6	GL2PetriNetEditor	DrawingEditor
7	GL2PetriSelectionCommand	SelectionCommand
8	GL2ConnectionSpline	SplineFigure
9	GL2PetriFixedText	TextFigure
10	GL2PetriTextFigure	TextFigure
11	GL2PetriTextFigureCommand	TextFigureCommand
12	GL2PetriTextSelectionCommand	TextFigureSelectionCommand

mas sua utilização do sistemas de restrições é pobre.

- GL2 apresenta a melhor utilização do sistema de restrições, o que lhes permitiu resolver facilmente alguns problemas acarretados por decisões erradas na especialização de classes
- GN apresenta problemas no aproveitamento de comportamento implementado pelo framework tanto no que diz respeito ao projeto de abstrações quanto ao sistema de restrições.

As tabelas 8.3 e 8.4 apresentam a lista de classes definidas pelos dois primeiros grupos, os quais utilizaram a ferramenta. A linhas ressaltadas nas tabelas indicam classes que implementam funcionalidade semelhante. A diferencia essencial entre as decisões de ambos grupos reside no fato do GL1 não ter projetado as figuras a serem editadas como figuras compostas, mas como figuras que são agrupadas através de um *GroupFigure* e restrições. O GL2, por sua vez, é cliente de figuras existentes no framework, retângulo e elipse, mostrando uma melhor decisão de projeto geral neste ponto, pois é possivel variar representação gráfica dos elementos do diagrama sem restruturar o código.

8.2.4.1 Comparação visual

A visualização de interações abstratas é de utilidade para verificar o grau de reutilização do framework obtido por diferentes aplicações. Através da comparação da interação das classes do framework nos diferentes níveis de abstração providos pela visualização é possivel detectar diferenças substanciais entre projetos diferentes. Se em um maior nível de abstração, são visualizadas só as classes que compõem o framework, existe uma sugestão de maior aproveitamento de funcionalidade, pois as classes da aplicação só estariam agregando comportamento específico, razão pela qual só aparecem nos níveis de detalhe maiores.

A FIGURA 8.4 apresenta uma análise comparativa das aplicações dos grupos GL2 e GN.

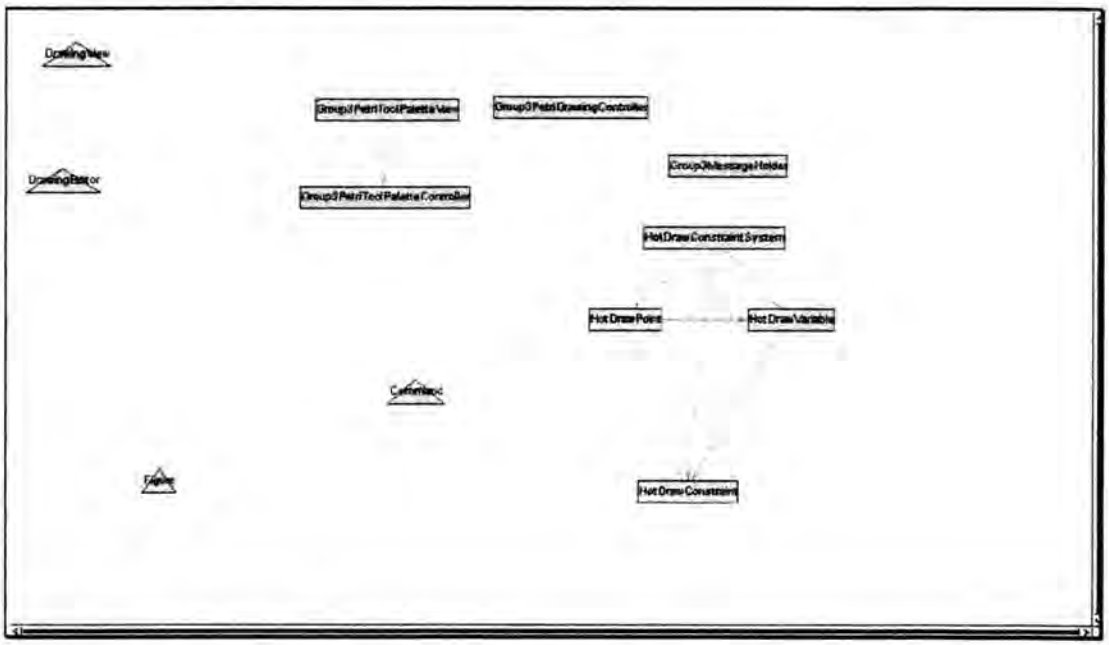


FIGURA 8.4- Visualização de hierarquia de alto nível dos editores GL1 e GN

Nesta figura é possivel observar os relacionamentos existentes entre as classes no nível de abstração topo de hierarquias, tendo-se utilizado cada editor para criar só uma Transição e um Estado. Neste nível de abstração (*topLevelHierarchy*), as classes destacadas em verde representam

classes do editor GN que reimplementam funcionalidade já existente no framework, a qual o GL1 reutiliza[4]. Por esta razão não aprecem classes do GL1 neste nível de detalhe. Esta funcionalidade refere-se, especificamente, ao comportamento relacionado com os controladores (*controllers*) do editor.

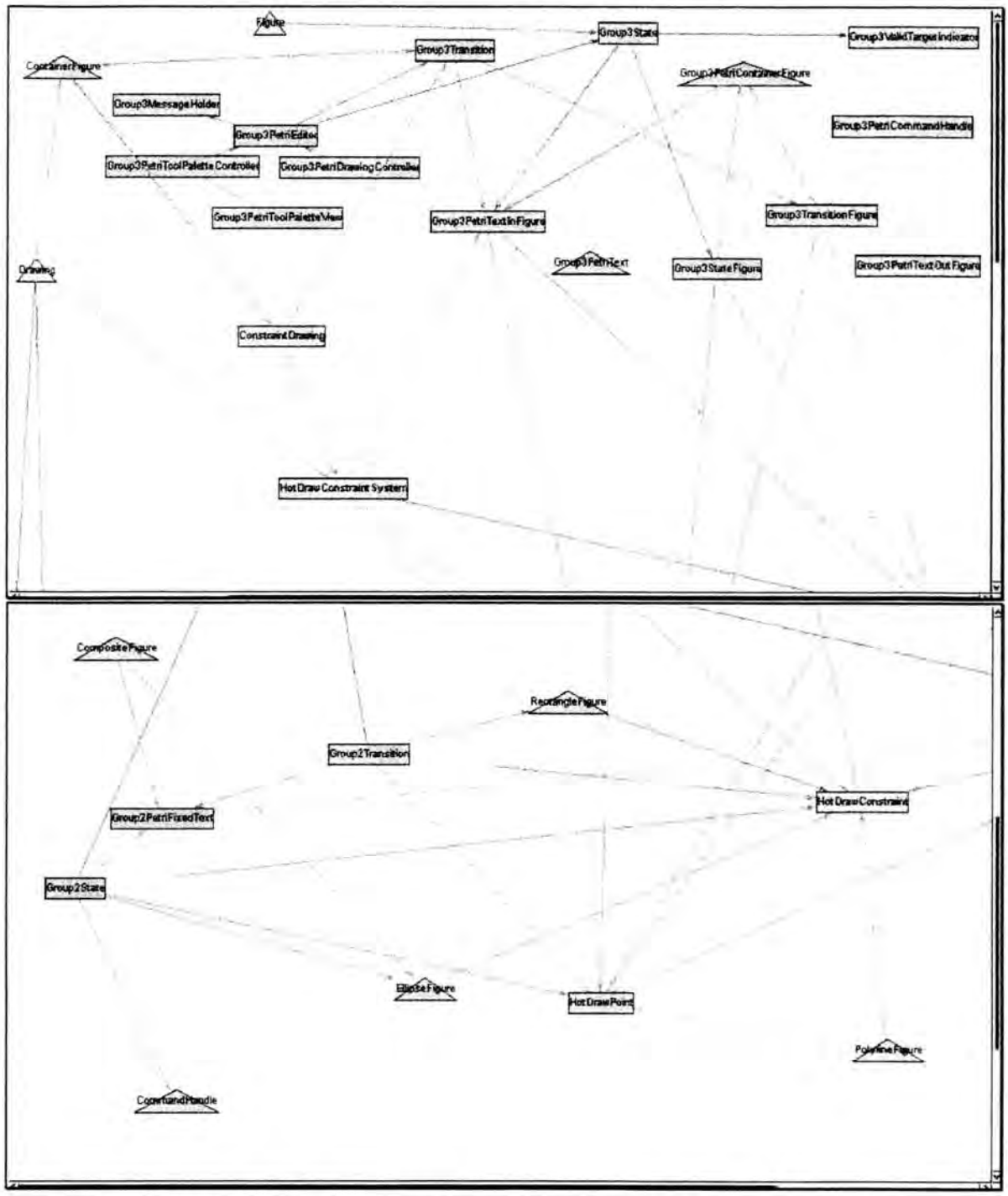


FIGURA 8.5- Máximo nível de detalhe dos relacionamentos entre as classes dos editores GL2 e GN

A FIGURA 8.5 mostra duas imagens dos relacionamentos no nível máximo de detalhe, ou seja, classes concretas [5]. Nelas é possivel observar que para a criação de duas figuras, no editor GN

existe maior relacionamento tanto entre as classes próprias do editor quanto entre as providas pelo framework (uma causa disto é a utilização de *ContainerFigure* como classe base das figuras e não *CompositeFigure*). O editor GL2, relaciona só duas classes, o qual indica um projeto claro, fácil de modificar e que obtém uma alta reutilização do framework.

Ambas figuras mostram sintomas evidentes de uma escassa compreensão do framework por parte do grupo GN, tanto no aspecto de reutilização de classes existentes, quanto a decisões de quais classes deveriam ser redefinidas e qual funcionalidade agregada.

8.2.4.2 Projeto das classes que implementam figuras

A decisão mais adequada para implementar a funcionalidade necessária para representar as figuras a serem editadas é herdar da classe *CompositeFigure*. Entre outras vantagens, herdar de *CompositeFigure* assegura o máximo aproveitamento da funcionalidade provida por *HotDraw*. Por exemplo, visualizar uma figura composta implica na visualização de cada um dos seus componentes, bem como deslocar uma figura implica na atualização automática da posição das figuras componentes.

Existe, entretanto, uma desvantagem derivada desta decisão de projeto, que é a pouca utilidade da implementação de área visível definida por *CompositeFigure*. Isto implica na necessidade de agregar comportamento específico para expandir uma figura cujos componentes devem ser expandidos como conseqüência da expansão da figura composta. Esta implementação não presume que a figura composta terá um desenho próprio, razão pela qual o seu tamanho é a soma dos tamanhos de seus componentes e não propaga mudanças no seu tamanho. Esta implementação responde ao critério que o desenho correspondente a uma figura composta será provido por um *wrapper* encarregado de acrescentar essa funcionalidade.

A FIGURA 8.6 apresenta uma visão parcial das classes definidas pelos editores GL1 (destacadas em laranja) e GN (destacadas em verde), para a mesma seqüência de execução de criação de uma transição e um estado em ambos editores. Como é simples observar o GL1 define estes componentes como subclasses de *CompositeFigure*, enquanto o GN implementa uma nova abstração equivalente com esta classe. Também pode observar-se que o GN cria uma classe que implementa o desenho da transição como subclasse de retângulo. GL1 optou por suas classes serem clientes de objetos existentes, elipse e retângulo.

Analisando detalhadamente a classe *Group3PetriContainerFigure*, é possível distinguir um conjunto de métodos:

```
figures:
new
setFigures:
do:
mainFigure:
figureAt:
handles
moveConstraint:
```

Estes métodos correspondem exatamente com o conjunto de métodos definidos pela classe *CompositeFigure*. Isto é, grande parte do comportamento provido pelo framework foi reimplementado.

O grupo GN argumentou que esta decisão foi baseada no problema da área visível, mas sua manutenção e atualização é simples através do uso do sistema de restrições de HotDraw ou através do mecanismo de anúncio de mudanças.

---

[5] A visualização foi separada e organizada espacialmente de forma tal de destacar as diferenças na imagem apresentada.

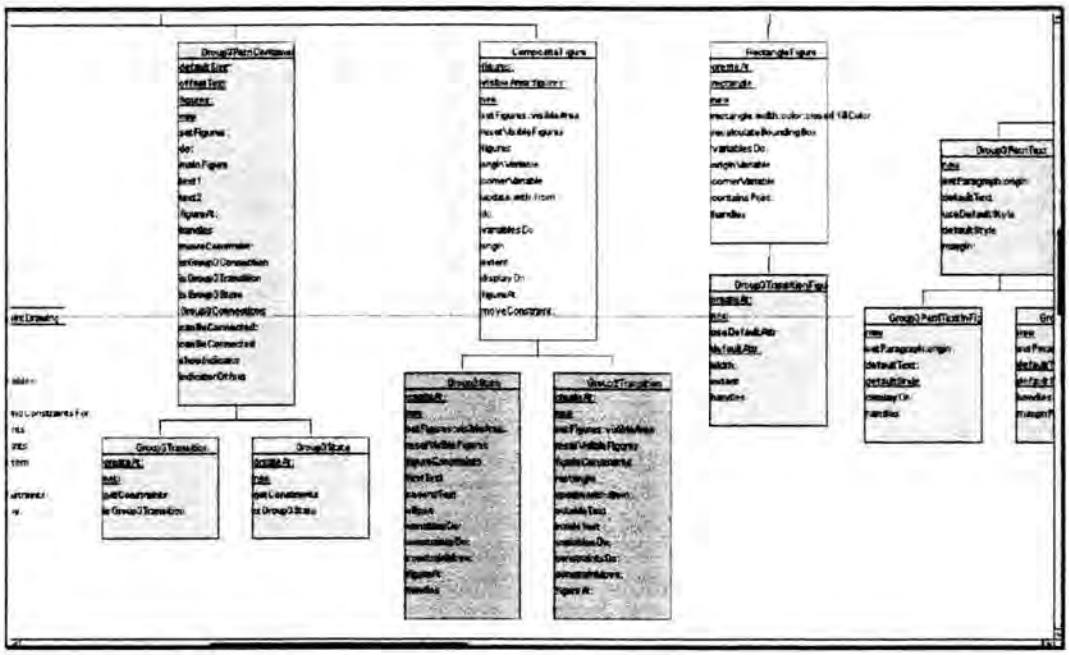


FIGURA 8.6- Visão Parcial de Classes de Figura definidas por GL1 e GN

No caso do editor GL1, este grupo define as classes que representam as figuras separadamente, isto é, não utilizaram uma abstração comum que represente o comportamento das figuras de uma Rede de Petri. A conseqüência desta implementação é a reimplementação dos mecanismos providos por *CompositeFigure*, e uma das principais causas das diferenças, detectadas pelas métricas, com o projeto do GL2.

Outro problema, importante de ser destacado, no projeto do GN é a utilização da hierarquia de *handlers*. Esta hierarquia define o comportamento dos pontos de seleção de uma figura utilizados para seu redimensionamento.

As subclasses criadas pelo GN são apresentadas na FIGURA 8.7, na qual pode observar-se a criação desnecessária de especializações que não acrescentam nem redefinem comportamento importante. Esta é uma clara sugestão de pouca compreensão do framework.

8.2.4.2.1 Conexões

Uma decisão de projeto correta tomada pelo GL1 é o manejo das conexões. Cada figura mantém duas listas, uma com os elos de entrada e outra com os de saída. O objeto representando a conexão mantém as referências à figura origem e a figura destino. Esta implementação favorece em muito a manutenção de consistência remoção de figuras.

- Ao remover-se uma conexão, esta conexão informa aos dois objetos que relaciona que deve ser removida das correspondentes listas que as figuras mantém de suas conexões.
- Quando uma figura é removida, causa a remoção de todas as suas conexões.

Os outros dois grupos, mantém a referência à suas conexões através da lista de dependentes. Esta solução também poderia ser boa se tivessem utilizado o mecanismo de anuncio de mudanças para informar às figuras relacionadas das remoções, mas, enviam mensagens diretamente a estas conexões, inibindo a uma figura de ter um dependente que não seja uma conexão.

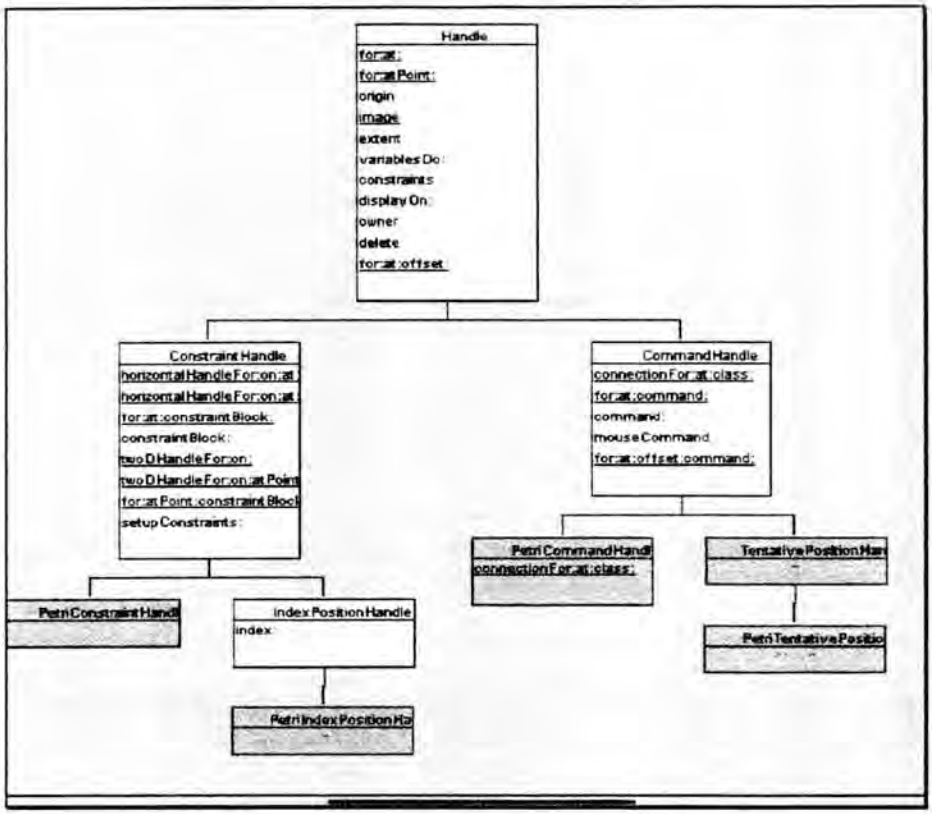


FIGURA 8.7- Classes de *handlers* criadas pelo GN

8.2.4.3 Manejo de restrições

O manejo das restrições é o ponto mais forte da implementação do GL1, mostrando um entendimento dos complexos protocolos envolvidos na sua utilização muito maior que o dos outros grupos. Este entendimento das restrições lhes permitiu implementar as figuras de forma mais independente uma das outras, evitando desta forma muitos dos conflitos que tiveram os outros dois grupos. Além disso, a melhor utilização das restrições lhes permitiu implementar a mesma funcionalidade utilizando um número menor de restrições.

8.2.5 Relação de tempos de desenvolvimento

Para ter uma idéia relativa dos tempos de desenvolvimento e o impacto da utilização da ferramenta, os grupos foram instruídos a registrar detalhadamente dias e horas durante as quais trabalharam até o final do desenvolvimento. A TABELA 8.5 apresenta o tempo aproximado empregado por cada grupo de acordo ao tempo dedicado exclusivamente a atividades de compreensão e o tempo dedicado ao desenvolvimento da aplicação.

Estes tempos apresentam dados muito interessantes que complementam e explicam

TABELA 8.5- Tabela de Tempos Empregados para Desenvolvimento

Grupo	Atividades de Compreensão(hs)	Desenvolvimento (hs)	Tempo Total (hs)	
GL1	72	100	172	
GL2	25	108	133	
GN	26	131	157	
Media/Desvio	41/26.8	113/16.1	154/19.6	

algumas das diferenças discutidas acima. Como é simples observar o grupo GL1 foi o que mais tempo dedicou a atividades de compreensão e o menor tempo de desenvolvimento, enquanto o GL2 foi o que empregou o menor tempo total. A diferença de tempos nas atividades de compreensão utilizando a ferramenta entre ambos grupos é a que marcou diferença entre a excelente utilização de sistemas de restrições do GL1 a respeito do GL2. O GL1, tendo utilizado a ferramenta pela segunda vez, optou por não começar o desenvolvimento até não alcançar uma compreensão total do framework, enquanto o GL2 a utilizou para alcançar uma compreensão global, e depois para analisar aspectos parciais que não podiam ser resolvidos durante a fase de programação. Esta estratégia foi a mesma utilizada pelo GN, o qual analisou os exemplos a partir do código, e podem-se perceber tempos semelhantes na atividade de compreensão.

O tempo dedicado pelo GL1 a atividades de compreensão os levou a concluir que utilizar *CompositeFigure* não era adequado pelo problema de atribuição de espaço aos componentes, e tendo compreendido bem como esta classe trabalha, optaram por replicar parte desse comportamento tomando em consideração a funcionalidade específica. O GL2, por sua vez, optou por replicar o projeto comum observado entre os exemplos analisados. Esta decisão é a mais adequada em termos de reutilização de funcionalidade, mas responde a critérios próprios de cada grupo de desenvolvimento. Neste aspecto, o GN tomou a mesma decisão que o GL1, mas o desenvolvimento de suas classes é totalmente diferente, replicando e rescrevendo código desnecessário por herdar de classes não adequadas.

8.3 Conclusões dos Experimentos

Dos experimentos realizados podem extrair-se algumas conclusões importantes acerca da utilização da ferramenta para a compreensão de frameworks, embora não definitivas, dada a magnitude da amostra que eles representam:

- A ferramenta ajudou para obter resultados muito melhores em termos de aproveitamento de funcionalidade provida pelos frameworks utilizados e qualidade final da solução.
- A ferramenta pode induzir uma exploração de detalhes que não necessariamente são relevantes. No entanto, a compreensão destes detalhes, ajudou ao GL1 para fazer uma utilização muito boa de um subsistema muito complexo, com é o sistema de restrições.
- A ferramenta não necessariamente ajuda para diminuir o tempo de desenvolvimento.
- O tempo total empregado pelo GL2 mostra que não existe um impacto importante de aprendizado da ferramenta.

Um aspecto que deve ser destacado é que em ambos experimentos os grupos que não utilizaram a ferramenta concentraram-se muito mais em aspectos externos das interfaces com o usuário que os grupos que utilizaram a ferramenta. Isto sugere que a utilização da ferramenta induziu estes grupos a se concentrarem muito mais nos aspectos de como reutilizar o framework que no desenvolvimento de funcionalidades específicas.

8.4 O Livro de Métricas

O livro de métricas é uma das aplicações mais interessantes de Luthier desenvolvidas, pois combina todas as funcionalidades providas pelo framework.

A coleta de métricas envolve tanto a exploração de informação estática quanto dinâmica. Esta informação é coletada por meta-objetos especialmente projetados para esta tarefa, utilizando para a coleta de informação estática os mecanismos já providos por Smalltalk. Esta

abordagem permitiu considerar aquelas classes envolvidas em um aplicação especifica, bem como todas as classes que compõem um framework separadamente.

A informação coletada é apresentada ao usuário através da geração de um livro que contém um capítulo para cada métrica com a informação coletada junto com a descrição do que essa métrica mede e um gráfico comparativo (FIGURA 8.8). Os totais das métricas são gerenciados por abstratores que controlam a contagem de acordo com três níveis de abstração: aplicação, classes, métodos. O nível de detalhe da visualização é controlado através desta escala, variando assim o gráfico apresentado bem como a informação de totais provida em outros paineis.

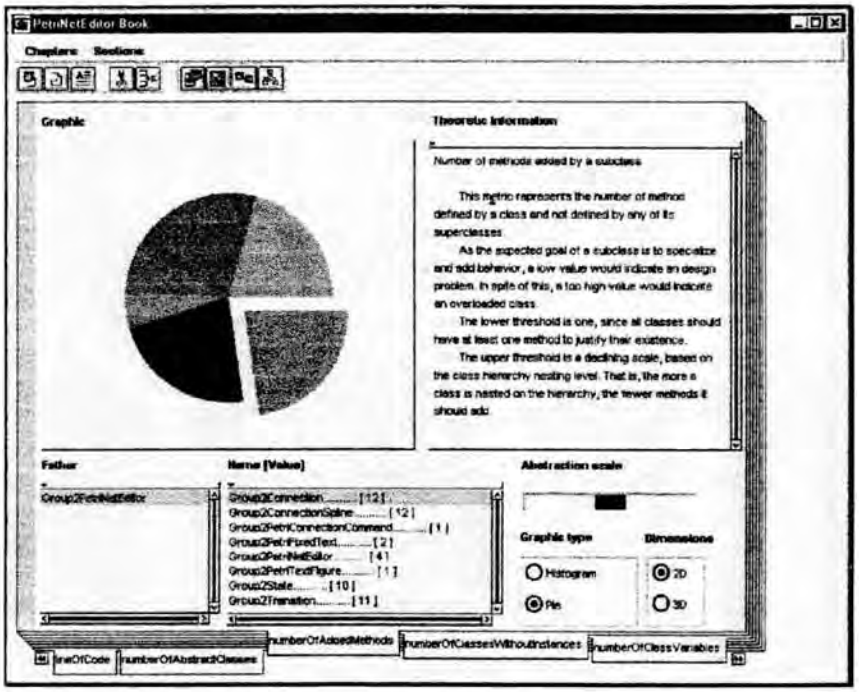


FIGURA 8.8- Número de métodos agregados por grupo utilizando Luthier

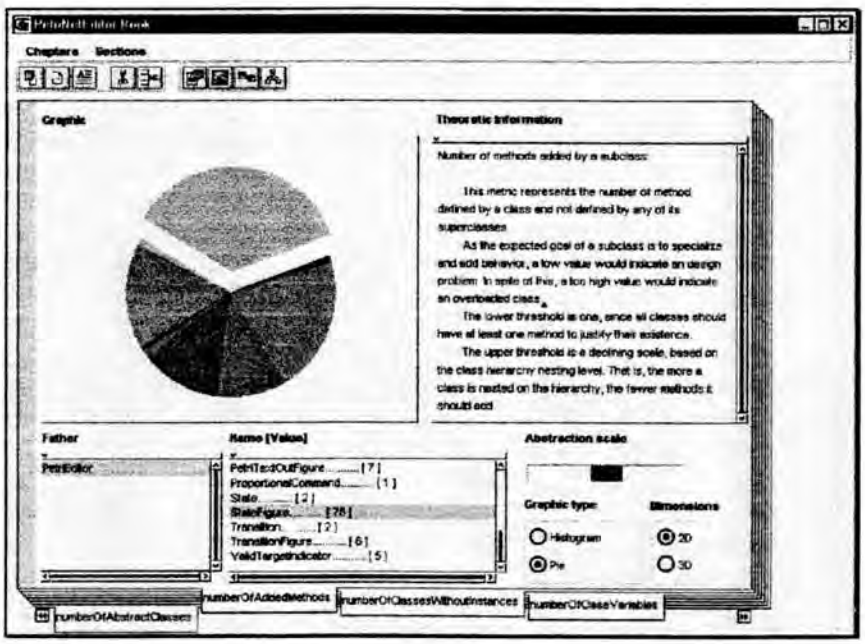


FIGURA 8. 9- Número de métodos agregados por grupo não utilizando Luthier

As FIGURAS 8.8 e 8.9 apresentam os resultados de aplicar a mesma métrica, numberOfAddedMethods, nos editores GL2 e GN. Para o cálculo destas métricas só se levaram em conta as classes específicas da aplicação e não as classes providas pelo framework.

Nos gráficos é simples observar que na primeira aplicação o número de classes é menor, sendo mais uniforme a distribuição dos métodos entre as distintas classes.

As figuras 8.10 e 8.11 mostram a comparação das mesmas aplicações através da métrica NumberOfOverriddenMethods. Neste exemplo, é possível observar a mesma relação entre as duas aplicações, isto é, os métodos da primeira tendem a estarem melhor distribuídos entre as classes.

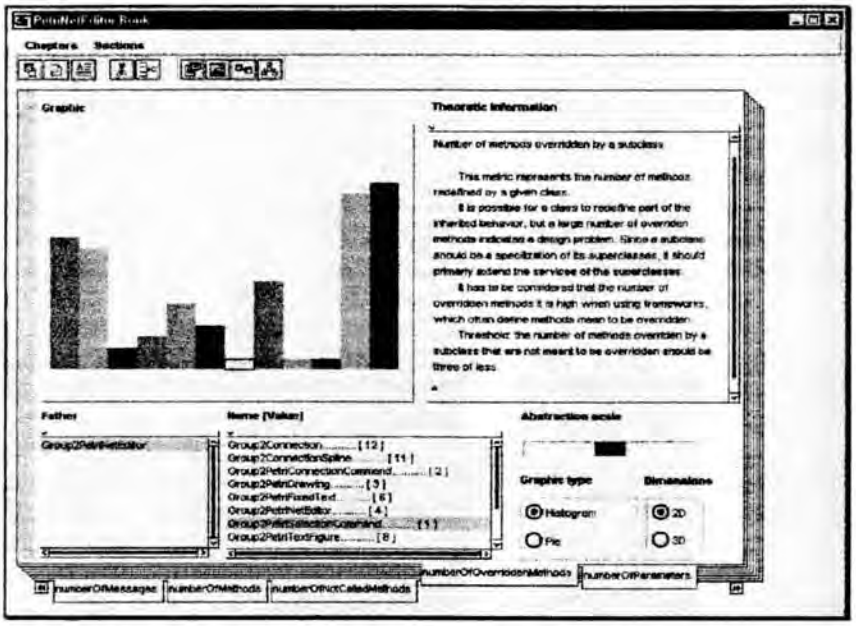


FIGURA 8. 10- Número de métodos redefinidos pelo GL2

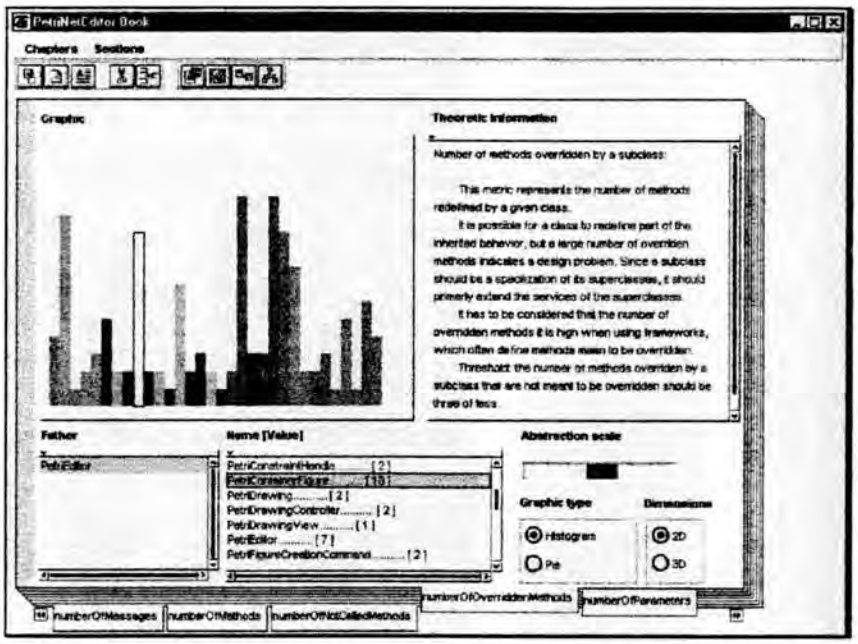


FIGURA 8. 11- Número de métodos redefinidos pelo GN

9 Conclusões

Nos capitulos precedentes, apresentou-se uma abordagem para a construção de ferramentas de apoio à compreensão de frameworks. Esta abordagem é suportada através de um framework, Luthier, que fornece a infra-estrutura adaptável para a construção de mecanismos de monitoração da execução de programas utilizando meta-objetos, reconhecimento de abstrações do programa analisado, e visualizações desse programa.

Com o suporte provido por Luthier uma ferramenta especialmente projetada para apoiar na compreensão de frameworks foi desenvolvida. Esta ferramenta permite a análise de aplicações construídas com um dado framework, coletando a informação relativa a esse framework utilizando uma representação de hipertexto. A partir desta representação, diferentes visualizações que enfatizam aspectos relevantes da estrutura do framework são geradas, através das quais o usuário pode analisar a estrutura do framework em diferentes níveis de abstração e orientar a análise até o nível de comportamento de instâncias.

Para testar a viabilidade e eficácia da abordagem, foram realizados experimentos na utilização de um framework. Estes experimentos sugerem um maior grau de reutilização do framework e de produção de aplicações de melhor qualidade, através da utilização da ferramenta de apoio.

A seguir discutem-se as contribuições principais realizadas pela tese, as limitações das abordagens adotadas, possíveis extensões e futuras pesquisas e, finalmente, algumas considerações acerca da eficácia da abordagem.

9.1 Resumo das contribuições

A tese apresenta várias contribuições nos diferentes aspectos abarcados pelo framework desenvolvido, que podem ser sumarizadas pelos seguintes pontos:

- A tese introduz uma estratégia de análise de programas orientada pela arquitetura, demonstrando praticamente como pode ser suportada através de uma ferramenta. Esta estratégia favorece, em um primeiro lugar, a organização da exploração da estrutura e comportamento de frameworks orientados a objetos, em termos de aspectos arquitetônicos de alto nível, evitando os problemas implementatorios e cognitivos que, freqüentemente, introduz uma análise centrada nas instâncias. Em conseqüência, a estratégia favorece a escalabilidade das ferramentas desenvolvidas para a análise de grandes aplicações.
- A tese demonstra a viabilidade da aplicação de técnicas reflexivas baseadas em meta-objetos para a análise de frameworks desenvolvidos em linguagens não reflexivas. Particularmente, o conceito de *gerenciadores de meta-objetos* introduzido, habilita a implementação de protocolos de meta-objetos especializados, impossíveis de serem desenvolvidos com as abordagens existentes na literatura para o suporte de meta-objetos em linguagens não reflexivas.
- Visualizações com zoom semântico baseado em *escalas de abstração*, externamente definíveis, é uma capacidade não encontrada nas ferramentas de visualização de software existentes na atualidade. Esta capacidade permite a exploração de um framework em diferentes níveis de abstração, evitando a proliferação de janelas, caso freqüente em outros sistemas de visualização, e, simultaneamente, simplificando a programação dessas visualizações.
- O conceito de *abstrator* representa uma estrutura de projeto que resolve de maneira elegante o problema de atribuição de responsabilidades entre os objetos que formam

uma ferramenta. Neste sentido, permite tanto orientar o projeto de novas ferramentas, bem como a adaptação dinâmica de ferramentas existentes, encapsulando funcionalidade de filtragem, seleção e construção de abstrações de forma totalmente independente das visualizações e da representação da informação. Esta caraterística representa um benefício significativo, pois aumenta o grau de reutilizabilidade de ambas partes citadas, evitando a criação de código específico de funções de análise de informação nessas partes e provendo, simultaneamente, o suporte para o gerenciamento automático do mecanismo de zoom semântico.

- A visualização de grafo de fluxo de mensagens, quando utilizada com volumes de informação razoáveis, oferece uma notação conveniente para suportar a animação arquitetônica de um dado framework, ressaltando aqueles métodos que devem ser implementados e mostrando, simultaneamente, exemplos de sua especialização e de seu papel nas interfaces definidas pelas classes abstratas.

- Os mecanismos de atribuição externa de semântica aos atributos visuais, tal como suportados por Luthier, não são facilmente encontrados nas ferramentas atuais de construção de visualizações. Estes mecanismos favorecem a construção de visualizações genéricas que podem ser adaptadas para usos específicos sem necessidade de programar comportamentos adicionais. Esta característica representa um ganho significativo de reutilização, promovendo um desenvolvimento muito mais independente das visualizações, que o possível através do suporte habitualmente provido pelos frameworks e ferramentas de visualização de software orientado a objetos hoje existentes.

- O suporte para o reconhecimento de abstrações é praticamente inexistente nas ferramentas visuais de apoio à compreensão de software orientado a objetos relatadas na literatura. O reconhecimento automático de padrões de projeto, embora restrito, e, essencialmente, a visualização do ponto de vista arquitetônico global, ressaltando aqueles métodos correspondentes a cada padrão dentro da hierarquia de classes, é uma abordagem que não foi praticamente explorada ainda na literatura.

- Frameworks para a construção de ferramentas de visualização das capacidades de Luthier não abundam na literatura. Neste sentido, Luthier representa uma contribuição importante para facilitar o desenvolvimento de outros projetos. Atualmente, está começando a ser utilizado por vários projetos de mestrado e doutorado no Instituto de Informática da UFRGS, bem como por vários projetos graduação na Universidade de Tandil, Argentina. Também está começando a ser utilizado na Universidade Autônoma de Madrid, España, como suporte para um projeto de doutorado sobre visualização de informação geográfica.

9.2 Limitações

A abordagem proposta nesta tese apresenta algumas limitações, tanto de índole conceitual quanto na atual implementação do framework.

A principal limitação de índole conceitual é a necessidade de recuperação de informação estática para orientar o início do processo de análise de uma aplicação. O início deste processo não é simples quando uma aplicação é desconhecida. Determinar as classes a serem refletidas requer de extensa análise estática das classes, para que o usuário decida quais classes devem ser inicialmente exploradas. Esta limitação, no entanto, também existe em qualquer ferramenta de análise de programas existente.

No caso de uma implementação C++, análise estática é necessária para implementar o mecanismo de reflexão das mensagens, mas uma vez que o programa foi instrumentado o comportamento do meta-nível pode servir tanto para a análise da aplicação quanto para estender sua funcionalidade. Deste modo a instrumentação de código para suporte de reflexão é mais justificada do que a instrumentação de código para geração de eventos, os quais só servem aos fins de análise de aplicações.

A implementação atual de Luthier ainda precisa de ser melhorada para poder ser distribuida a usuários em geral. Essencialmente, muitas classes, desenvolvidas para versões anteriores, ainda precisam ser modificadas para corresponder completamente com o projeto final do framework relatado no capítulo 6. Além disto, esta implementação apresenta as limitações descritas a seguir. A seguir discutem-se as principais limitações e possíveis formas de serem resolvidas.

9.2.1 Análise de sistemas distribuídos e concorrentes

A atual implementação da ferramenta de análise de frameworks não suporta a análise de sistemas distribuídos nem concorrentes. Esta limitação, é devida em princípio à natureza de centralizada de Smalltalk que não facilitou esta experimentação. Entretanto, adaptar a ferramenta para a análise de aplicações distribuídas é simplificado pela utilização de meta-objetos acrescentados transparentemente às aplicações. Estes meta-objetos podem monitorar o comportamento de objetos rodando em máquinas diferentes e comunicar a informação coletada a um gerenciador da informação localizado em outra máquina.

O caso de sistemas concorrentes não é suportado, pois Smalltalk suporta muito limitados a nível de blocos, razão pela qual não é possível implementar uma concorrência real de aplicações. Uma alternativa, para dotar a ferramenta com este suporte, pode ser aprimorar a coleta de informação para tomar em consideração o *thread* de execução corrente através da análise da pilha de execução.

9.2.2 Dependência da linguagem

Algumas das soluções implementadas são dependentes do suporte provido pelo sistema Smalltalk, o qual limita um pouco sua implementação direta em outras linguagens. Esta limitação, entretanto, pode ser facilmente ultrapassada com mecanismos específicos programados para as outras linguagens.

A implementação de abstratores, tal como descrita neste trabalho, não pode ser facilmente portada a C++ ou Java, por exemplo, devido à natureza tipada dessas linguagens e a falta de um mecanismo automático de propagação das mensagens. Este mecanismo de propagação faz com os abstratores possam ser transparentes às visões e serem tratados como objetos da representação.

Para o caso de C++, por exemplo, seria necessário definir um protocolo fixo e o mecanismo de escalas de abstração ser incorporado através de herança múltipla. Isto solucionaria, em parte, estas limitações, mas continua sendo necessário programar abstratores específicos para cada tipo de objeto da apresentação que será visualizado em forma abstrata.

Esta limitação, no entanto, não diminui a importância do conceito aplicado como princípio sistemático de distribuição de funcionalidade para o projeto de ferramentas adaptáveis.

Na implementação de Luthier em C++ o suporte de análise de instâncias e a configuração dinâmica de meta-objetos, requer que todas as classes que compõem a aplicação sejam pre-processadas, bem como as bibliotecas que essa aplicação utiliza, pois é necessário que sejam compiladas em forma conjunta. Caso isto não for realizado, um estratégia mais dinâmica, como a mostrada para Smalltalk, requer da regeneração de toda a aplicação para cada nova classe a ser analisada. Isto, evidentemente limita a aplicabilidade da abordagem, mas não inibe sua utilidade.

9.2.3 Eficiência

A implementação dos mecanismos de abstratores e regeneração de visões é ineficiente quando a quantidade de informação coletada é muito grande. Isto faz com que as visualizações não possam ser regeneradas em um tempo de resposta que habilite um efeito continuo de *zooming*.

O reconhecimento de padrões de projeto é uma tarefa muito demorada, devido, essencialmente, a ineficiência inerente do interpretador Prolog utilizado, o qual é programa no próprio Smalltalk. Este duplo nivel de interpretação impõe um desempenho relativamente pobre. A implementação das regras, por outra parte, também é extremamente ineficiente, sendo uma outra fonte da ineficiência da função. Esta ineficiência, no entanto, pode ser tolerada se considera-se a utilidade da informação provida pelos padrões.

9.2.4 Mecanismos de consulta

A implementação atual do mecanismo de consultas é altamente dependente do conhecimento que o usuário possua de detalhes da representação da informação. Este mecanismo deve ser melhorado em uma futura versão da ferramenta, para permitir consultas textuais mais gerais, independentes da sintaxe Smalltalk e o envio direto de mensagens à representação interna das mensagens.

9.3 Extensões e Áreas de Pesquisa Futura

Luthier apresenta muitas possibilidades de ser estendido para prover um apoio mais avançado tanto para a compreensão de frameworks quanto para o suporte de projeto de frameworks. Nesta seção descrevem-se algumas das possiveis extensões e linhas de pesquisa que poderiam ser seguidas a partir da base provida por Luthier.

9.3.1 Geração de ferramentas de apoio à instanciação de aplicações

Normalmente um framework é acompanhado por uma biblioteca de componentes prontos para serem utilizados. Se os componentes adequados estão prontos em uma biblioteca é possível gerar aplicações simplesmente compondo estes componentes sem necessidade de programar novos. Por exemplo utilizando o framework *Model-View-Controller* de Smalltalk, a interface de uma aplicação consiste de instâncias de *controllers* e *views* conectadas e parametrizadas com menus e mensagens a serem enviados em resposta a certos eventos. Neste caso, praticamente é possível compor a interface de uma aplicação sem necessidade de programar novas classes, sendo em geral suficientes as classes definidas na biblioteca.

Esta composição pode ser realizada através de um editor gráfico de manipulação direta que apresente visualmente os componentes e suporte sua conexão, garantindo a semântica da composição. Existem várias ferramentas desenvolvidas para o framework MVC, que suportam a composição interativa de aplicações através de manipulação direta das componentes definidas pelo framework, como por exemplo Glazier [ALE 87], Alternate Reality Kit [SMI 87], VisualBuilder do ambiente VisualWorks [PAR 94], ou VisualAge [IBM 93], entre os mais relevantes.

Estes editores suportam a composição visual da interface de uma aplicação através da manipulação direta da representação visual das instâncias que implementam cada componente de interface. Em alguns casos, como VisualAge por exemplo, a interface pode ser visualmente conectada com a representação icônica de componentes parametrizados existentes em uma biblioteca, segundo uma interface padronizada. Este enfoque não acrescenta nenhuma capacidade adicional de instanciação já que os componentes devem ser programados segundo padrões estabelecidos pelo framework. Isto implica no problema básico da compreensão do framework.

A construção deste tipo de editores, obviamente, depende de cada framework em particular, para realmente serem de utilidade na instanciação de qualquer aplicação do domínio. Uma aplicação interessante de ser pesquisada é extensão de Luthier, para apoiar na geração de ferramentas

visuais de instanciação. A informação coletada pela ferramenta de análise, permite obter um modelo do framework, no qual, os relacionamentos válidos entre componentes podem ser utilizados para guiar um editor visual. Este editor pode orientar o usuário acerca de quais componentes existem e quais são relacionamentos válidos entre esses componentes, bem como, facilitar o acesso ao código, naqueles casos em que for necessário criar novo comportamento. As capacidades do sub-framework de interfaces são adequadas para permitir a construção de editores com mínimo esforço, os quais podem até utilizar as visualizações já existentes na biblioteca.

9.3.1.1 Assistência inteligente

Uma área interessante de pesquisa é a aplicação de técnicas de inteligência artificial para orientar o processo de utilização de um framework. Particularmente, a pesquisa atualmente sendo desenvolvida na área de agentes inteligentes pode ser de utilidade enorme para o desenvolvimento de agentes especializados no apoio na instanciação de frameworks.

Um agente é uma entidade autônoma que possui conhecimento acerca de uma dada tarefa e capacidades comportamentais que lhe permitem decidir ou planejar como aplicar esse conhecimento para desenvolver uma tarefa ou resolver um dado problema.

Através de uso de conhecimento acerca do domínio e a forma em que um framework resolve os problemas desse domínio, um agente poderia orientar o usuário acerca de quais as partes do framework que deveriam ser utilizadas ou especializadas para obter a funcionalidade desejada. A capacidade de planejamento que algumas arquiteturas de agentes fornecem, por exemplo, permitiria pensar na possibilidade de gerar planos de instanciação, que descrevam as ações que deveriam ser seguidas para satisfazer o objetivo de construir uma aplicação com determinados requisitos. Uns dos problemas a ser pesquisados, entre outros, é como expressar o conjunto de conhecimento que o agente deve possuir acerca do framework, para gerar tal plano de instanciação.

Esta capacidade, combinada com a geração de editores de instanciação, permitiria a geração de ambientes visuais inteligentes que poderiam facilitar em muito a utilização de um dado framework, aproveitando as vantagens que as duas tecnologias podem oferecer.

9.3.2 Apoio ao desenvolvimento de frameworks

O desenvolvimento de framework é habitualmente uma atividade iterativa que envolve várias iterações de um projeto original. Em cada iteração as classes existentes são refatoradas para aumentar sua reutilizabilidade, criando superclasses abstratas de conjuntos de classes concretas, acrescentando métodos e variáveis ou relocalizando métodos e variáveis em classes diferentes, entre outras operações.

A través do suporte de análise dos exemplos, Luthier pode ser estendido para apoiar na manipulação e realização manual das refatorações. Também, refatorações semi-automáticas, ao estilo do *Software Refactory*, podem ser implementadas. Para isto informação estática detalhada da estrutura de métodos é necessária. Esta informação pode ser obtida na forma de árvores de reconhecimento diretamente do compilador, o qual está integrado com o ambiente de desenvolvimento Smalltalk. Baseado nesta informação várias refatorações podem ser automaticamente sugeridas e materializadas no código sob controle do projetista.

Uma área mais promissora ainda é a provisão de assistência automatizada ao desenvolvimento baseado em padrões de projeto. A organização da informação em termos de livros, habilita a criação de catálogos de padrões, tanto arquitetônicos como de padrões de projeto, os quais poderiam ser utilizados para apoiar na derivação do projeto inicial de um framework.

Nesta linha, vários aspectos estão ainda em aberto, como por exemplo, a seleção de padrões de um catálogo baseada em especificações funcionais de alto nível acerca do problema a ser resolvido. Um outro problema a ser pesquisado é a provisão de suporte para a aplicação de um dado padrão selecionado a um projeto em desenvolvimento e a transformação automática da estrutura desse projeto de acordo com as classes e métodos que o padrão prescreve.

Ferramentas de apoio ao projeto desta natureza representariam um avance substancial no estado da arte em ferramentas de apoio ao projeto de software orientados a objetos. Neste sentido, a abordagem orientada a regras adotada para representação dos padrões, poderia ser pesquisada como uma alternativa para suportar atividades desta natureza.

9.3.3 Técnicas avançadas de visualização, realidade virtual e documentação na WWW

A utilização de técnicas de visualização tridimensional é ainda uma área em aberto para visualização de software. Técnicas de visualização e realidade virtual podem representar um avance significativo para a compreensão de software orientado a objetos e especialmente de frameworks e sistemas reflexivos. Por exemplo, vários modelos arquitetónicos podem ser caraterizados naturalmente por estruturas tridimensionais, como por exemplo arquiteturas de niveis ou arquiteturas distribuidas.

O desenvolvimento das extensões de HTML, VRML e VRML2, abrem possibilidades importantes para a navegação e exploração não inmersiva em mundos virtuais distribuídos. A utilização desta tecnologia para a exploração de bibliotecas de software, e particularmente de frameworks orientados a objetos, é uma área que não tem sido ainda explorada. Estas capacidades poderiam permitir, por exemplo, suportar um processo de compreensão cooperativo entre diferentes grupos distribuidos geograficamente utilizando um dado framework para desenvolver aplicações.

O suporte de visualização provido por Luthier pode ser estendido para gerar representações HTML e em conseqüência, VRML, da informação coletada, bem como browsers para estas linguagens podem ser implementadas em Smalltalk, oferecendo a possibilidade de múltiplas máquinas virtuais interconectadas através do WWW, e compartilhando todas as capacidades do ambiente para o desenvolvimento de software.

Do ponto de vista da documentação de um framework, Luthier pode ser empregado para apoiar na geração de documentação pelos projetistas. Esta documentação pode então estar disponivel para seu acesso através da rede por usuários do framework.

9.3.4 Conhecimento de domínio e atribuição de conceitos

A falta de informação ou conhecimento acerca do domínio do framework analisado restringe a informação aportada pelas ferramentas a aspectos estruturais do projeto de um framework. O conhecimento de domínio, utilizado em ferramentas de engenharia reversa, permite suportar um processo de *atribuição de conceitos* às construções codificadas pelo framework. Isto permite explorar o programa analisado do ponto de vista da semântica das abstrações codificadas em termos do domínio de aplicação. Definitivamente, um suporte desta natureza seria de grande ajuda para facilitar a compreensão de um dado framework.

Por exemplo, se a ferramenta possuisse conhecimento acerca do modelo MVC e fosse capaz de reconhecer que o framework analisado corresponde com esta estrutura, então seria possível oferecer explicações em termos mais abstratos da forma em que as partes do framework colaboram e o papel que cada uma dessas partes tem dentro do projeto global. Sem este conhecimento, o suporte a ser provido fica restrito a facilidades para explorar e analisar a informação do framework, baseando-se exclusivamente em aspectos estruturais de granularidade fina, como por exemplo tipos de métodos.

A incorporação de conhecimento de domínio não depende dos mecanismos de captura de informação, nem da representação dessa informação, sendo um problema de recuperação de abstrações de muito mais alto nivel que simples abstrações estruturais. Neste sentido, a incorporação de regras que permitam explicar as abstrações de um framework, em termos da semântica do domínio é um aspecto que pode ser explorado para aumentar as capacidades fornecidas por Luthier para facilitar a compreensão de um framework dado.

9.3.5 Reengenharia de sistemas orientados a objetos

Atualmente existe uma tendência crescente nas empresas e organizações, em paises como Brasil e Argentina, no interesse da incorporação de técnicas de orientação a objetos para o

desenvolvimento de software. Em muitos casos, experiências protótipo com pequenos sistemas estão sendo desenvolvidas, em outros, sistemas completamente novos estão sendo desenvolvidos utilizando orientação a objetos (particularmente em Visual C++). Assim, é razoável supor uma migração paulatina à incorporação integral de orientação a objetos como técnica de desenvolvimento predominante.

Neste contexto, é necessário considerar que o desenvolvimento de software orientado a objetos de boa qualidade não é uma tarefa que possa ser efetuada facilmente por programadores inexperientes na tecnologia. Particularmente, o projeto de tais sistemas é, definitivamente, mais complexo de ser realizado do que com a tecnologia procedimental convencional. Uma tendência geral entre os iniciados na tecnologia é, por exemplo, a sobreutilização de herança que conduz a programas inflexíveis e difíceis de ser adaptados sem uma grande proliferação de classes.

Tendo em consideração a alta mobilidade de pessoal que carateriza algumas organizações, esta situação pode levar ao conhecido problema de manutenção de sistemas legados, mas com a complexidade adicional, já discutida, que os programas orientados a objetos apresentam para sua compreensão. Desta forma, a reengenharia de sistemas orientados a objetos, uma área pouco explorada ainda, pode tornar-se um dos problemas centrais da reengenharia dentro de poucos anos.

Neste sentido Luthier apresenta possibilidades interessantes de ser explorado e estendido para a construção de ferramentas de suporte à reengenharia de sistemas orientados a objetos. Através das possibilidades de manipulação da estrutura das classes, é possível acrescentar funcionalidade de edição que permita para manipular diretamente a estrutura de atributos e métodos. Esta funcionalidade é complementar com a necessária para apoio ao desenvolvimento de frameworks.

9.4 Considerações Finais

Observando a evolução atual das técnicas de desenvolvimento de software, a orientação a objetos parece estar entrando na etapa de consolidar-se como tecnologia padrão para o desenvolvimento de software dentro das organizações. Nesta linha de evolução, é razoável supor que a produção de frameworks seja o objetivo principal desses desenvolvimentos. Entretanto, é necessário tomar em consideração que os frameworks só representam *uma tecnologia de implementação adequada* para aumentar a produtividade no desenvolvimento, mas não uma panacéia para os problemas do desenvolvimento de software.

O desenvolvimento de frameworks é uma tarefa que requer, além de experiência no domínio de aplicação, um grande nível de conhecimento de estruturas de projeto que permitam produzir soluções efetivamente reutilizáveis. Produzir um framework, considerado como estável e maduro, é ainda uma tarefa que requer de um investimento substancial de tempo e esforço. Assim, a questão de quando é justificado ou não o desenvolvimento de um framework é uma questão de índole econômica, na qual o custo de desenvolvimento só pode ser justificado em termos da *qualidade* e *produtividade*.

. A abordagem apresentada nesta tese, representa um passo importante no estudo dos problemas envolvidos com a *compreensão* e *utilização* de frameworks. O grau no qual ferramentas de apoio podem contribuir para a tornar mais fácil e eficiente o processo de compreensão de frameworks depende, em grande medida, não só das capacidades fornecidas por essas ferramentas, mas também das capacidades cognitivas e intelectuais de cada usuário da ferramenta. Das experimentações realizadas, pode extrair-se como conclusão parcial, mas muito importante, que a ferramenta não ajudou por si mesma para diminuir os tempos de desenvolvimento das aplicações, mas ajudou a produzir aplicações que estejam de acordo com a estrutura do framework utilizado, e com um número bastante menor de classes específicas desenvolvidas. Tomando em consideração que a funcionalidade das aplicações desenvolvidas foi exatamente a mesma, pode-se concluir, então, que o uso da ferramenta de apoio contribuiu para a produção de aplicações com maior grau de reutilização e, portanto, de qualidade e manutenibilidade maior. Este resultado é muito mais importante, no longo prazo, que um ganho relativo em termos de tempos de desenvolvimento.

A tese não abordou o problema do projeto de frameworks. Este problema é, certamente, um dos aspectos essenciais que define a utilidade de um framework. No entanto, a abordagem apresentada pode resultar de grande ajuda para apoiar no processo de análise de exemplos para a

refatoração do framework, mantendo, simultaneamente, a documentação atualizada. Esta capacidade é de grande importância para reduzir os tempos de desenvolvimento.

Certamente, maior pesquisa na linha introduzida pela tese, ajudará para tornar a aplicação de frameworks, em escala industrial, uma realidade que efetivamente possa contribuir para melhorar tanto a produtividade e qualidade final do software produzido quanto a manutenção e adaptação desse software.

Anexo 1

Notação de Contratos Utilizada para a Especificação de Luthier

Neste anexo apresenta-se a descrição da notação utilizada para especificação de Luthier. Esta técnica é utilizada porque é a que melhor permite descrever o projeto detalhado do framework, tanto no nível de responsabilidades de classes quanto no nível de organização de instâncias.

1.1 Contratos

A técnica de contratos (*contracts*) [HEL 90] é uma técnica textual especialmente desenvolvida para documentar frameworks, do ponto de vista dos relacionamentos de colaboração entre classes e objetos que um framework predefine.

Um contrato descreve as colaborações que existem entre um conjunto de participantes, através de uma linguagem semi-formal. Com esta linguagem especificam-se os participantes na relação e suas obrigações contratuais, como variáveis que devem possuir, a interface externa que devem prover e a seqüência ordenada de ações que devem instanciar certas pós-condições como verdadeiras em resposta às mensagens.

1.2 Símbolos, Operadores e Expressões

A notação se baseia essencialmente nos operadores clássicos de lógica de primeira ordem e conjuntos, estendidos com alguns operadores especiais para expressar envio de mensagens, e seqüenciamento de operações:

$\varnothing$	conjunto vazio
$\in$	pertence
$\notin$	não pertence
=	atribuição
$\Rightarrow$	implica
$\Leftrightarrow$	se e só se
$\vee$	ou
$\wedge$	e
$\forall$	para todo
$\exists$	existe
$!\exists$	não existe
||	operador paralelo
;	operador seqüencial
Δ *variável*	especificador de possível mudança de estado da variável associada
{	precedência de ordenamento

Pre e pós-condições: {*predicados*}

Mensagens: receptor → operação(argumentos)

Clausuras:

Aplicação de um conjunto de *ações* a todos os elementos de um conjunto descrito por um *predicado*, unidas pelo *operador*, ou seja, aplicação quantificada, paralela ou seqüencial.

<*operador variável* : *predicado* : *ações*>

1.2.1 Extensões realizadas

O formalismo base descrito foi aumentado com alguns operadores e funções que permitem expressar com maior precisão a semântica de operações abstratas e das pre e pós-condições.

◊ (predicado | ação)	Este operador é utilizado para expressar ações que deveriam ser executadas por métodos abstratos ou hook, mais que não podem ser predeterminadas por uma especificação de pre e pós-condições
ϕ	Método vazio. É utilizado para especificar métodos cuja implementação não é relevante para o contrato
current	Representa o objeto corrente
former(variável)	Representa o valor anterior de uma variável depois de uma atribuição a essa variável.
ref(objeto)	Representa a referência interna a um objeto. Esta função é utilizada para especificar quando as referências a dos objetos são trocadas.
type(objeto)	Representa o tipo ou classe do objeto
perform(operação)	Executa uma operação de um objeto
intercepted selector	Representa qualquer método interceptado por algum mecanismo de interceptação
presents argument	Expressa que uma visão apresenta visualmente os valores do seu argumento. Habitualmente o argumento é um modelo.

1.3 Definição de contratos

Um contrato é definido de acordo ao seguinte formato geral:

```
CONTRACT <nome do contrato>
    <Participante P1> SUPPORTS:[

        <serviço>(<argumentos) ⇒ <especificação de pre-condições>
                                 <especificação de ações>
                                 <especificação de pós condições>
        .....
    ]

    <Participantes P2> SET(P2) WHERE EACH P2 SUPPORTS:[
    ....

    ]
    <Participante Pn> SUPPORTS:[
                ...
    ]
INVARIANT
    <invariante>
INSTANTIATION
    <ações de instanciação>
END CONTRACT
```

O contrato está definido por um conjunto de participantes cujas obrigações são definidas pela cláusula **SUPPORTS**. Um participante pode representar um conjunto de um dado tipo de obrigações, definido pela clausula **SET(<Tipo>) WHERE EACH <Tipo> SUPPORTS**. Isto

habilita a descrição da composição de comportamento de conjuntos de objetos que interagem segundo um mesmo protocolo.

Cada serviço está constituído por o nome, o nome e tipo dos argumentos que recebe, e uma seqüência de condições e ações.

Opcionalmente, é possivel definir um invariante que seus participantes do contrato colaboram para manter.

Também, opcionalmente, é possivel descrever como são instanciados os participantes do contrato, em termos de combinações de instâncias.

Por exemplo, o contrato ModelView abaixo, define dois participantes *Model* e *Views*. *Views* é um conjunto de componentes do tipo *View*, cada um dos quais suporta os serviços *setModel* (que indica para um *View* qual é o seu modelo) e *refresh* que tem como pre-condição que a visão tem que refletir o estado do modelo para executar a operação *display*. O modelo, por sua vez, vai indicar para cada visão associada quando mudou o seu estado, através da mensagem *changed*.

```
CONTRACT ModelView
 Model  SUPPORTS:[
    // conjunto de atributos
    atributes []
        // atribuição a variável
    setVarValue(varName,val) ⇒ Δ atributes[varName].value;
                                    // invoca ao serviço changed   para informar a mudança
                                    changed()
                                    // pós-condição: a variável fica com o valor val
                                    {atributes[varName].value=val};
        // Seqüencialmente cada View recebe a mensagem refresh
    changed() ⇒ <;v: v  Views: v →refresh() >

    addView(v) ⇒{ v ∈  Views }

    removeView(v) ⇒{ v ∉ Views }
    ]

 Views : SET (View) WHERE EACH  View SUPPORTS [

        // atualiza a apresentação na tela depois de atualizar os valores
    refresh() ⇒ display(){
                View presents  Model.atributes}.
    display()⇒ ϕ

    setModel(m)⇒{ model=m }
    ]

 INVARIANT
        // Cada visão sempre mostra o estado atual do seu modelo
    Model.setVarValue(varName,val)
        < ∀ v: v  Views: v presents Model.atributes>
INSTANTATION
        Model→addView(View)
        View→setModel(Model)
END CONTRACT // ModelView
```

1.4 Definição de Classes e Clausulas de Conformação

Uma vez definidos os contratos, é necessário definir como os serviços fornecidos pelas classes do framework conformam as obrigações definidas pelos participantes. Uma declaração de conformação requer especificar como cada serviço definido por um participante é substituído por um método dessa classe, caso possuam nomes diferentes.

A definição do mapeamento é estendida para definir quais os métodos base e *template*, quais métodos poderiam ser redefinidos e quais métodos devem ser implementados por subclasses. cada método deve especificar seu role na classe através de um comentário informal.

Na notação utilizada neste texto, uma classe é definida com o formato seguinte:

```
CLASS <Classe>
    INHERITS FROM <Superclasse>
    CONFORMS <Participante> IN <Nome Contrato>
    Classe SUPPORTS:[

        attributes
            <atributo>:Tipo
                role: <comentário>
        base methods
            <nome de serviço> [ as <nome de método>]
                role: <comentário>
            ...
        template methods
            <nome de serviço> [ as <nome de método>]
                role: <comentário>
            ...
        subclass might redefine
            <nome de serviço> [ as <nome de método> ]
                role: <comentário>
            ...
        requires subclass to support:
            <nome de serviço> [ as <nome de método> ]
                role: <comentário>
]
CONFORMS <Participante> IN <NomeContrato 2>
        <Classe>SUPPORTS:[
]
END CLASS <Classe>
```

Anexo 2

Catálogo de Padrões de Projeto

Os padrões de projeto são padrões de organização de hierarquias de classes, protocolos e distribuição de responsabilidades entre classes, que caraterizam construções elementares de projeto orientado a objetos. Este apêndice oferece uma descrição sintética das caracteristicas essenciais e da estrutura de classes dos padrões do catálogo [GAM 94].

A2.1 Padrões do Escopo de Classes

A jurisdição de classes abrange aqueles padrões que prescrevem relacionamentos estáticos entre classes e suas subclasses. Essencialmente estes relacionamentos consistem na organização de funcionalidades em termos da herança e o estabelecimento de protocolos internos entre classes e subclasses.

Em geral os padrões desta jurisdição baseiam-se em classes abstratas que definem algum protocolo, o qual é implementado ou completado por subclasses. Os padrões dentro deste escopo se classificam em três categorias:

- Padrões Criacionais: Permitem tornar independente a forma em que os objetos são instanciados dos detalhes do processo de criação. Estes padrões utilizam herança para variar a classe de objeto a ser criado pelo código que é herdado do seu padre.
- Padrões Estruturais: Utilizam herança para compor protocolos ou código. O exemplo mais simples é a utilização de herança múltipla para compor uma interface uniforme partindo de duas ou mais classes. O resultado é uma classe que amalgama a semântica das classes base.
- Padrões Comportamentais: Capturam a forma em que classes cooperam com as suas subclasses para completar sua semântica.

A2.1.1 Factory Method (Criacional)

Tipicamente uma classe base define métodos, *NewObject* da FIGURA A2.1 por exemplo, para criar e manipular instâncias de alguma outra classe (*ProductClass*), mas esta última classe depende de cada aplicação particular. Para desacoplar esta dependência, a classe base define um método abstrato, digamos *CreateInstance*, o qual é invocado pelo método *NewObject*. O método *CreateInstance* será definido por cada subclasse para criar instâncias de classes especificas.

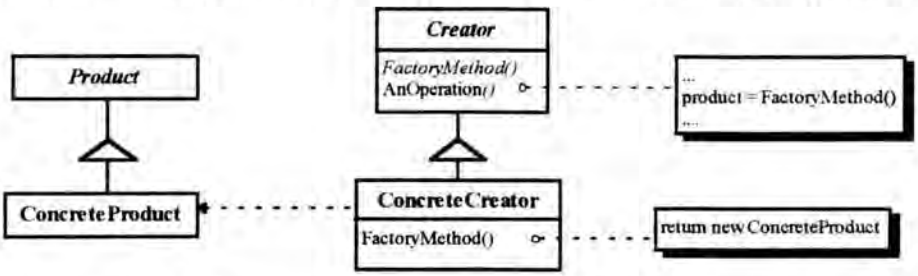


FIGURA A2.1- Estrutura abstrata de classes do padrão *Factory Method*

A2.1.2 Adapter (Estrutural)

Converte a interface de uma classe existente *(Adaptee)* a uma interface esperada pelos clientes de um determinado tipo de objetos *(Target)*, permitindo tratar uniformemente classes desenvolvidas independentemente. Classes existentes que não poderiam ser utilizadas são adaptadas por uma nova classe *(Adapter)* que traduz as requisições de serviços num protocolo ao protocolo da classe adaptada, através de herança múltipla.

A complexidade desta tradução pode variar desde uma simples troca de nomes de serviços até combinações complexas de um conjunto de serviços da classe adaptada.

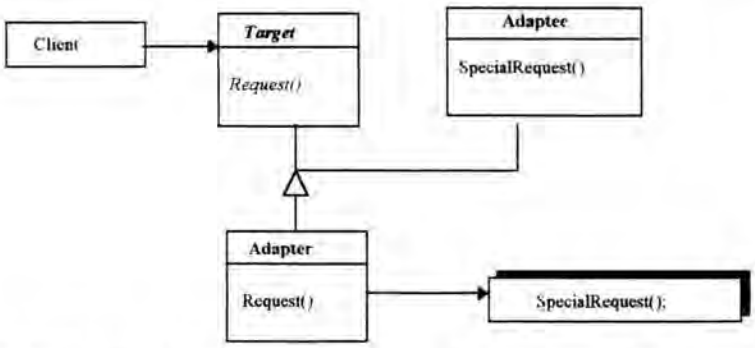


FIGURA A2.2- Estrutura abstrata de classes do padrão *Adapter*

A2.1.3 Template Method(Comportamental)

Um método *template* define o esqueleto de um algoritmo deixando alguns passos do algoritmo serem definidos por subclasses. As subclasses devem implementar o comportamento especifico para prover os serviços requeridos pelo algoritmo

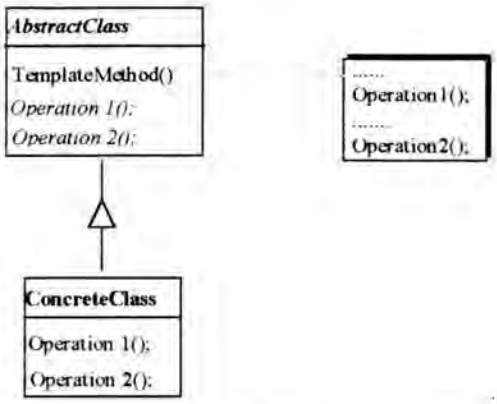


FIGURA A2.3- Estrutura abstrata de classes do padrão *Template Method*

A2.1.4 Interpreter(Estrutural)

Define como representar a gramática, árvore de sintaxe abstrata e interpretador para linguagens simples, em termos de uma hierarquia de classes. Cada elemento da gramática define uma classe, a qual encapsula o código para interpretar a expressão.

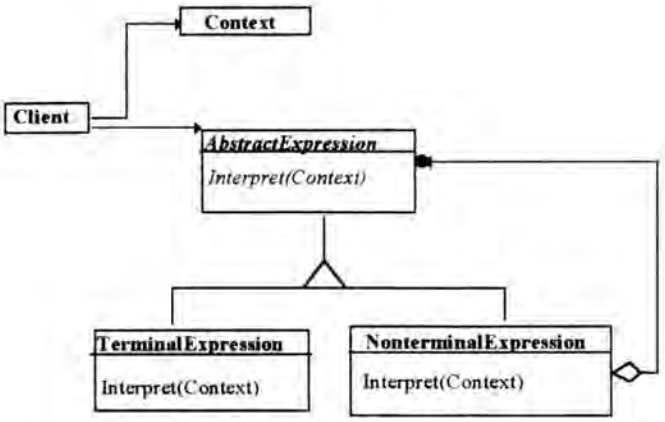


FIGURA A2.4- Estrutura abstrata de classes do padrão *Interpreter*

A2.2 Padrões do Escopo de Objetos

Os padrões de objetos aplicam-se todos a várias formas de composição de objetos não recursiva. Composição representa a forma mais poderosa de reutilização, os objetos são mais facilmente reutilizáveis através de variações na forma na qual são compostos que através da forma em que se organizam numa hierarquia de subclasses. Também são divididos de acordo as categorias anteriores:

- Padrões Criacionais: Permitem que um cliente possa criar objetos através de um protocolo estabelecido por uma classe, mas que interagindo com o conjunto de classes específicas que definem os objetos. Basicamente, abstraem a forma na qual conjuntos de objetos são criados, delegando a criação de um objeto em outro objeto, fornecendo aos clientes uma interface comum para criar diferentes tipos de objetos. Os diferentes padrões nesta categoria diferenciam-se no processo utilizado para criação das instâncias e na utilização de herança. A diferença com os criacionais de classes reside em que estes últimos definem relacionamentos entre subclasses, enquanto os criacionais de objetos definem relacionamentos entre múltiplas classes.
- Padrões Estruturais: Descrevem formas de compor objetos para implementar novas funcionalidades. A flexibilidade que este tipo de padrões acrescenta vem da possibilidade de mudar a composição dinamicamente, o qual é impossível com composição estática de classes.
- Padrões Comportamentais: Envolvem a distribuição de algoritmos e responsabilidades entre grupos de objetos. Essencialmente, eles definem complexos padrões de colaboração entre objetos que freqüentemente se materializam como seqüências de controle muito complexas dentro da aplicação. Compreender este fluxo de controle é geralmente uma tarefa difícil, porém, compreendendo o padrão de comportamento só é necessário compreender a interconexão de objetos.

A2.2.1 Abstract Factory (Criacional)

Define uma interface para criar diferentes tipos de objetos sem conhecer as classes específicas utilizadas para sua criação. Os clientes interagem com uma única classe para criar objetos específicos através de mensagens (*MakeProductA* por exemplo). Estas mensagens são definidas por uma classe abstrata. Subclasses desta classe abstrata implementam cada método para criar instâncias de classes específicas, as quais são retornadas ao cliente.

As classes específicas são organizadas em hierarquias independentes. Deste modo é possível variar dinamicamente o tipo de objetos produto criados por um cliente, sem necessidade de alterar o código deste último.

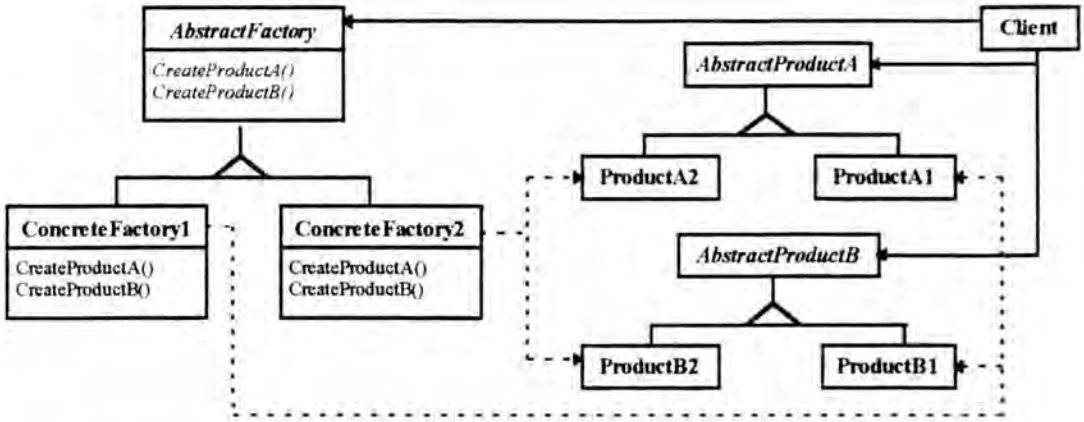


FIGURA A2.5- Estrutura abstrata de classes do padrão *Abstract Factory*

A2.2.2 Builder (Criacional)

Separa a construção de um objeto complexo da sua representação de modo tal que o mesmo processo de construção pode criar diferentes representações. Neste caso o processo de construção é controlado por outro objeto (*Controller*), o qual envia mensagens ao objeto *builder*. O protocolo utilizado pelo cliente de um builder é definido por uma classe abstrata. Cada builder específico traduzirá estas mensagens (*BuildPart*) na representação que ele encapsula, construindo incrementalmente o objeto produto. Finalizado o processo de construção o objeto produto é retornado ao cliente.

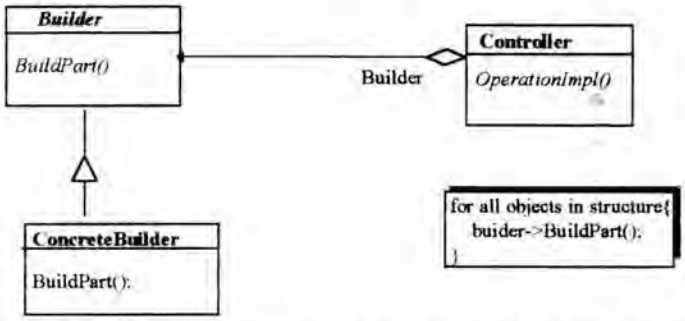


FIGURA A2.6- Estrutura abstrata de classes do padrão *Builder*

A2.2.3 Prototype (Criacional)

Especifica os tipos de objetos a serem criados utilizando uma instância prototipica como especificação e criando novos objetos copiando o protótipo. Isto permite que os objetos a serem criados possam ser definidos dinamicamente. Esta última caraterística o faz o mais flexível dos padrões criacionais. A través dele é possivel, também, incorporar classes à aplicação que são carregadas em tempo de execução e que não foram vinculadas com a aplicação razão pela qual não é possivel invocar o construtor.

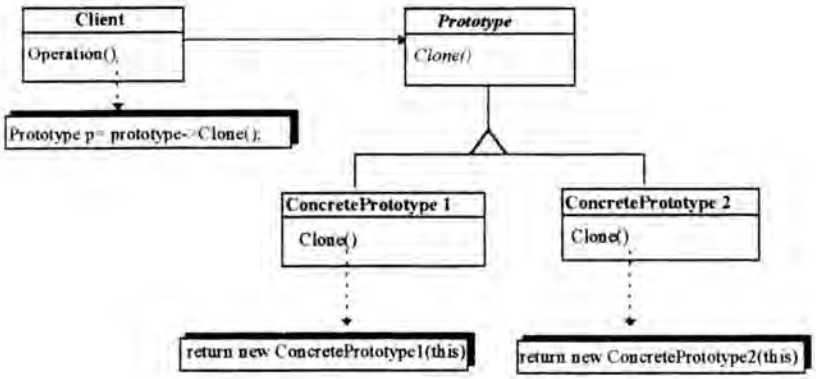


FIGURA A2.7- Estrutura abstrata de classes do padrão *Prototype*

A2.2.4 Singleton (Criacional)

Define uma classe que terá só uma instância acessivel por toda a aplicação. Os clientes interagem com a classe, a qual mantém uma referência à instância criada e não permite a criação de novas instâncias. Deste modo evita-se a utilização de variáveis globais para fazer globalmente acessivel um conjunto de serviços.

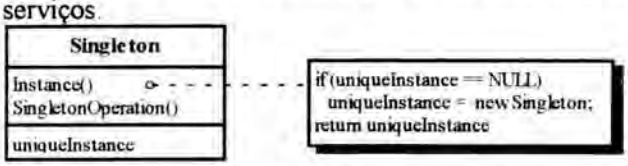


FIGURA A2.8- Estrutura abstrata de classes do padrão *Singleton*

A2.2.5 Adapter (Estrutural)

O *adapter* de objetos utiliza composição e não herança para adaptar as interfaces de classes existentes. Neste caso, o *adapter* mantém uma referência ao objeto adaptado ao qual delega as requisições de serviços realizadas pelos clientes.

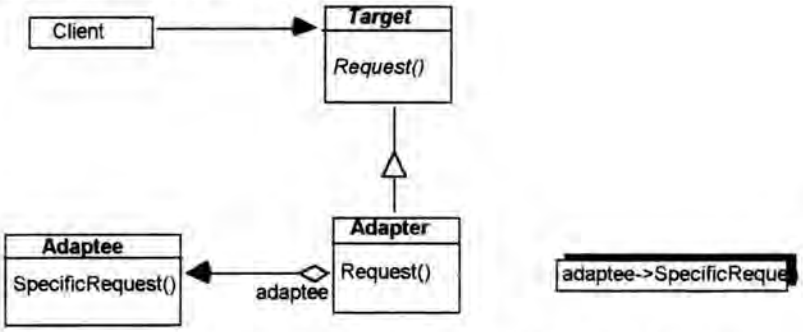


FIGURA A2.9- Estrutura abstrata de classes do padrão *Adapter*

A2.2.6 Bridge (Estrutural)

Suporta variações da implementação de uma abstração representando a interface e a implementação com classes separadas. Esta separação permite associar dinamicamente a implementação desejada para uma abstração. Mesmo assim, é possivel compartilhar uma implementação entre múltiplos objetos.

O mecanismo básico definido pelo padrão é a passagem das mensagens recebidas pela abstração ao implementador que ela referencia. No entanto, tipicamente a implementação define operações de baixo nível, as quais são compostas pela abstração para fornecer operações de maior nível de abstração.

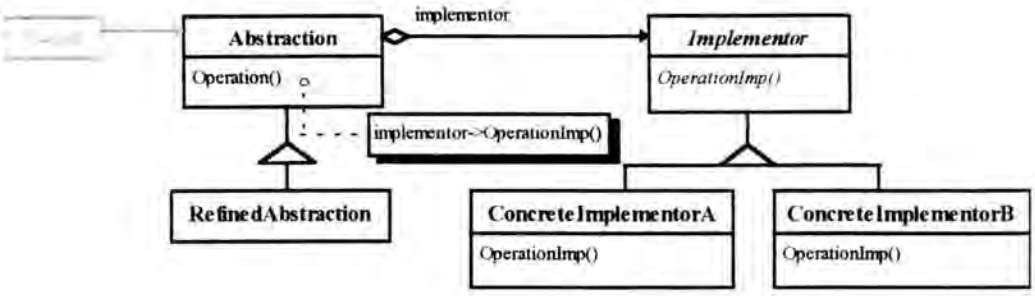


FIGURA A2.10- Estrutura abstrata de classes do padrão *Bridge*

A2.2.7 Composite (Estrutural)

Compõe objetos em estruturas de árvore para representar hierarquias de partes ou de conteúdo. O padrão permite tratar múltiplos objetos compostos recursivamente como um objeto simples, simplificando assim o código dos clientes. O padrão diferencia os compostos (*Composite*) das folhas (*Leaf*), sendo uma superclasse de ambos (*Component*) a que representa o objeto composto. Tipicamente o objeto composto distribui a aplicação de operações entre seus filhos, obtendo uniformidade através do polimorfismo.

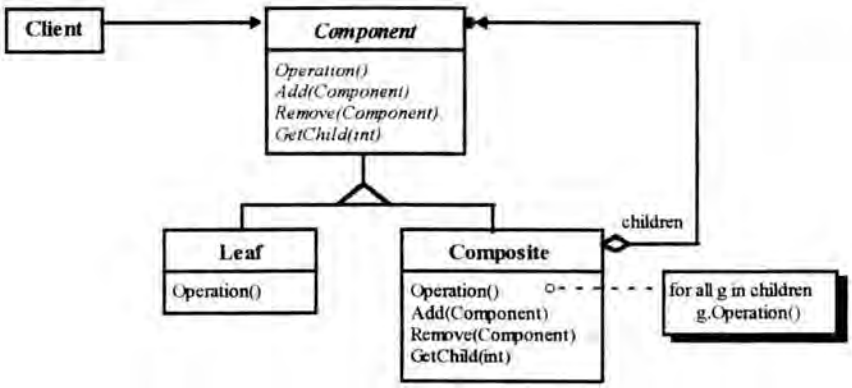


FIGURA A2.11- Estrutura abstrata de classes do padrão *Composite*

A2.2.8 Decorator (Estrutural)

Acrescenta dinamicamente serviços adicionais, propriedades ou comportamento a objetos através de composição. Vários *Decorators* podem ser compostos recursivamente para acrescentar múltiplas propriedades a um objeto, evitando a criação de subclasses específicas. Basicamente um *decorator* realiza alguma função específica (*AddedBehavior*) e transfere o controle ao seu componente invocando a mesma operação. O *Decorator* é um *Composite* degenerado, no sentido que restringe a composição a um componente só.

Através dele é possível acrescentar a objetos caraterísticas ortogonais recursivamente, sem que o objeto em questão tenha conhecimento e evitando herdar estas propriedades para estarem disponíveis. O exemplo típico é a decoração do marco de uma janela, a qual é independente do conceito de janela e pode até não existir. Se a janela fosse responsável de criar seu marco, cada decoração diferente obrigaria a criar subclasses novas da janela para implementar cada apresentação. Utilizando um *decorador* estas caraterísticas podem ser variadas dinamicamente.

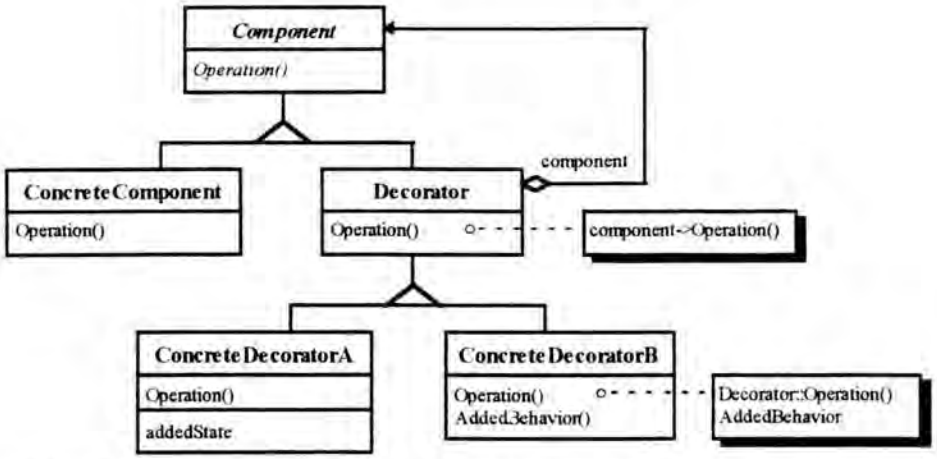


FIGURA A2.12- Estrutura de abstrata de classes do padrão *Decorator*

A2.2.9 Façade (Estrutural)

Define um ponto de entrada único para acessar objetos dentro de um subsistema, fornecendo um nível de encapsulamento maior para estes objetos. Os clientes interagem com os recursos de um subsistema através de uma interface comum, a qual encapsula a versão específica do subsistema sendo utilizada. Obviamente, isto reduz as dependências entre subsistemas incrementando a reutilizabilidade e simplificando o código dos clientes.

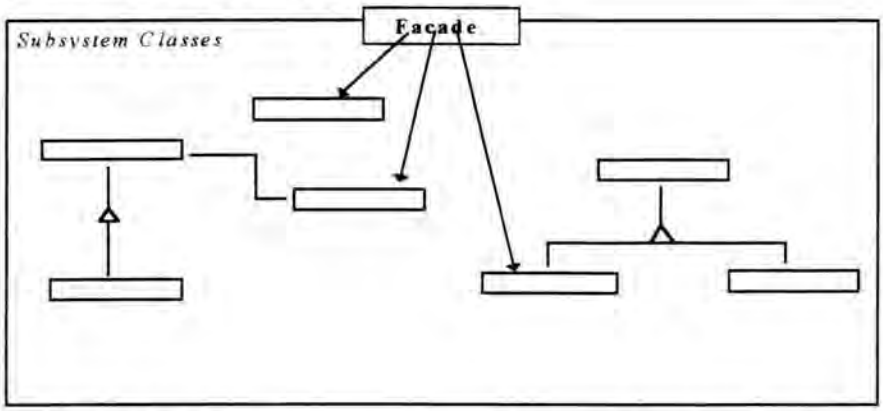


FIGURA A2.13- Estrutura abstrata de classes do padrão *Façade*

A2.2.10 Flyweight (Estrutural)

Define como objetos podem ser compartilhados eficientemente, suportando abstração de objetos no nível mais baixo de granulosidade (caracteres por exemplo).

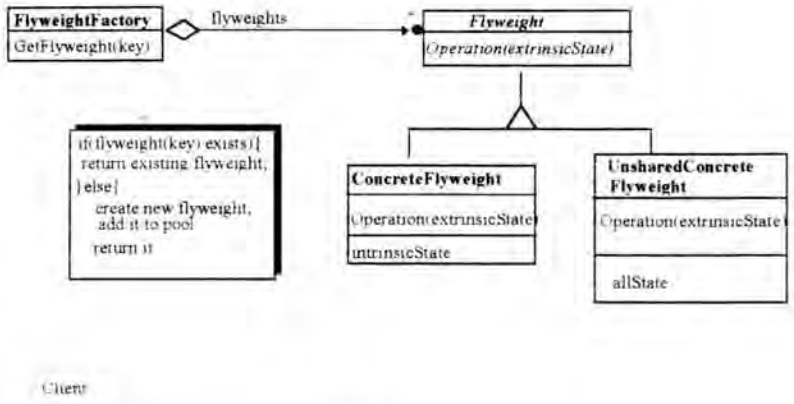


FIGURA A2.14- Estrutura abstrata de classes do padrão *FlyWeight*

A2.2.11 Proxy (Estrutural)

Age como um intermediário de outro objeto, podendo restringir, aumentar ou alterar as propriedades de um objeto.

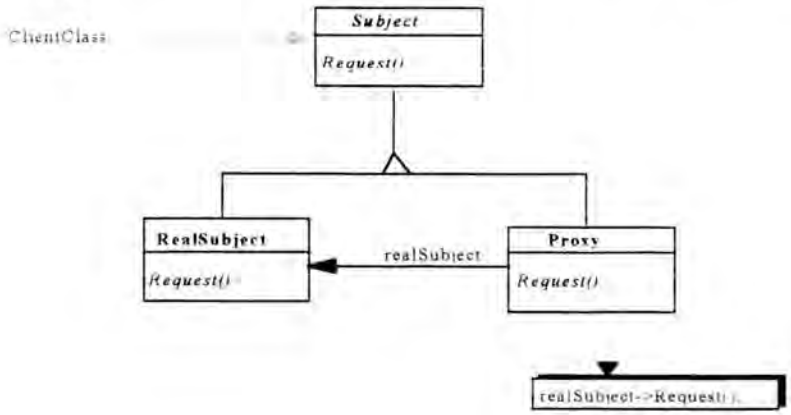


FIGURA A2.15- Estrutura abstrata de classes do padrão *Proxy*

A2.2.12 Chain of Responsability (Comportamental)

Define uma hierarquia de objetos, tipicamente organizados desde o mais especifico até o mais geral, os quais tem a responsabilidade de manejar uma requisição. A requisição é passada

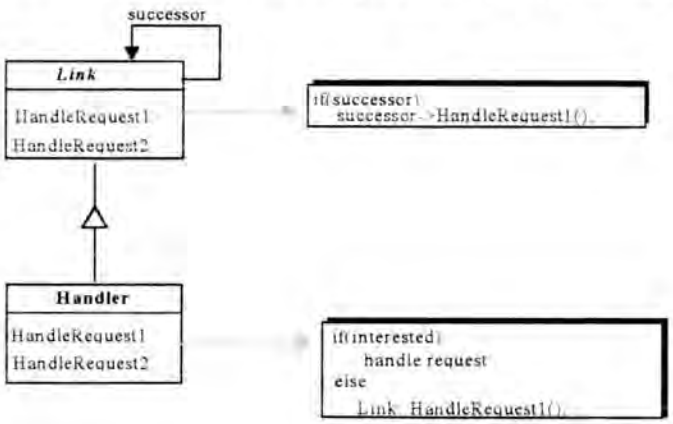


FIGURA A2.16- Estrutura abstrata de classes do padrão *Chain of Responsability*

A2.2.13 Command (Comportamental)

Este padrão encapsula num objeto uma requisição de um serviço. Através dele é possivel desacoplar o criador da requisição do serviço do executor desse serviço. O mecanismo de comandos é tipico nas aplicações interativas, nas quais o usuário invoca serviços fornecidos pela aplicação, através de algum componente de interface, como menus por exemplo. Separando o inicio da requisição do processo da mesma é possivel manter informação acerca das mudanças produzidas pela execução do comando para suportar a volta atrás (*undo*) destas mudanças. Da mesma forma é possivel pospor a execução da operação ou mudar dinamicamente o alvo do comando.

Na estrutura básica, uma classe abstrata *Command* define o protocolo de execução (*Execute*) que será invocado pelo iniciador da requisição. Cada subclasse de Command implementará este serviço invocando à ação correspondente do seu alvo. Uma caraterística adicional é que os comandos podem ser compostos para formarem *macros*.

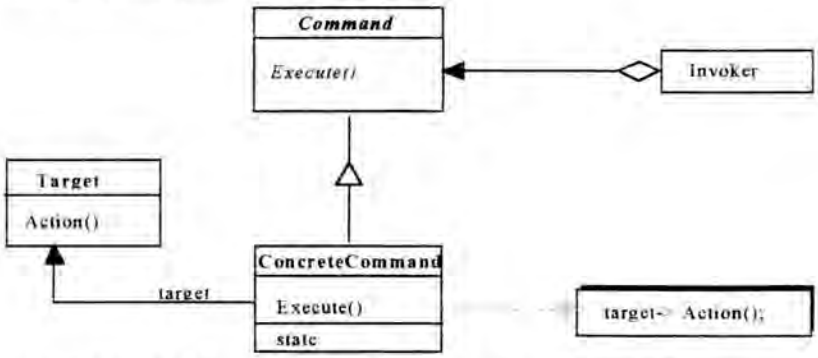


FIGURA A2.17- Estrutura de abstrata de classes do padrão *Command*

A2.2.14 Iterator (Comportamental)

O *Iterator* tem por objetivo encapsular em objetos algoritmos para percorrer estruturas de dados sem expor a representação interna. Uma classe especial (*Iterator*) define a interface abstrata para percorrer estruturas seqüencialmente. As estruturas estão definidas em outras classes, as quais devem fornecer o mecanismo para criar um iterador do tipo adequado sobre elas (*ConcreteIterator*).

A separação do algoritmo permite definir diferentes estratégias para percorrer a estrutura mantendo sempre a mesma interface, definindo diferentes subclasses do *Iterator*. Isto apresenta duas vantagens: os clientes sempre vão a interagir utilizando o mesmo conjunto de mensagens para percorrer a estrutura e esta última não precisa ser contaminada com comportamentos que podem depender da aplicação.

Dependendo da linguagem de programação nem sempre é necessário manter uma hierarquia de classes separada. Smalltalk, por exemplo, fornece blocos os quais podem ser utilizados para parametrizar um iterador general no momento da sua criação.

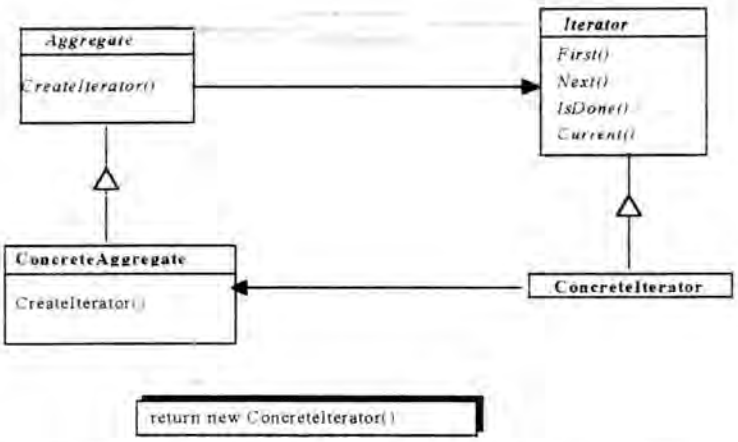


FIGURA A2.18- Estrutura abstrata de classes do padrão *Iterator*

A2.2.15 Mediator (Comportamental)

O *Mediator* coordena a comunicação entre objetos que interagem, localizando a complexidade da interação em outra classe. Essencialmente, o padrão cria um mediador das comunicações de modo tal que os restantes objetos só interagem com ele e não entre eles diretamente. Quando os objetos comunicados encapsulam funcionalidade altamente reutilizável em diferentes contextos, a utilização do padrão evita que estes objetos possuam conhecimento de outros que podem depender de cada aplicação particular e não corresponde a nenhum deles implementar ou controlar a interação.

O caso típico são os componentes de interface com o usuário, como botões, listas, etc.; os quais podem compor diálogos de diferente natureza. Incorporando uma classe *DialogDirector* por exemplo, que coordene o dialogo, os componentes só devem se comunicar com ela.

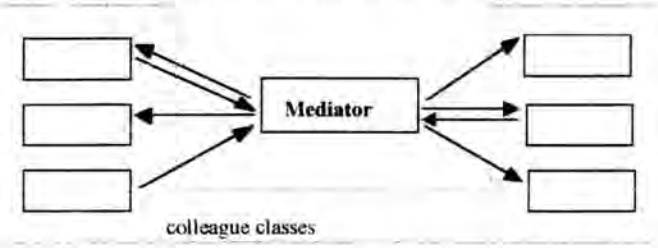


FIGURA A2.19- Estrutura abstrata de classes do padrão *Mediator*

A2.2.16 Memento (Comportamental)

Um *memento* é um objeto que armazena o estado interno de outro objeto. Este estado interno é posteriormente utilizado para restabelecer o objeto original ao seu estado prévio. O objeto origem *(Originator)* do memento tem a responsabilidade de criar um memento quando este é requerido, evitando assim fornecer uma interface para acessar o seu estado interno. Isto evita expor desnecessariamente detalhes de implementação, segurando o encapsulamento. Idealmente o memento só mostra uma interface restrita ao objeto encarregado de manter o estado anterior (*Caretaker*), evitando seu acesso interno (sempre que a linguagem o suporte); e uma interface ampla ao originador para poder restabelecer o estado armazenado.

A principal aplicabilidade dos mementos é em conjunção com comandos (ver *Command*) que tem a capacidade de voltar atrás operações efetuadas sobre outros objetos.

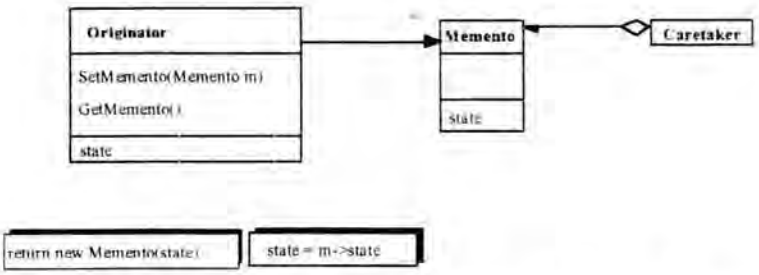


FIGURA A2.20- Estrutura abstrata de classes do padrão *Memento*

A2.2.17 Observer (Comportamental)

O *Observer* preserva restrições de sincronização, coordenação e consistência entre objetos. Quando um objeto muda seu estado, todos os objetos dependentes dele são notificados e atualizados automaticamente.

O padrão é classicamente utilizado para manter consistentes as possivelmente múltiplas apresentações gráficas de um objeto da aplicação. Entretanto também pode ser aplicado para manter consistente qualquer conjunto de objetos que apresentam dependências mútuas, mas que não conhecem que depende deles.

Essencialmente o padrão define um mecanismo de comunicação de eventos. Estes eventos devem ser sinalizados pelo objeto observado (*Subject*) quando se produz alguma mudança de interesse no seu estado interno. Os objetos dependentes (*Observers*) devem receber a notificação desta mudança e interagir com o *subject* para obter a informação necessária para atualizar seu próprio estado.

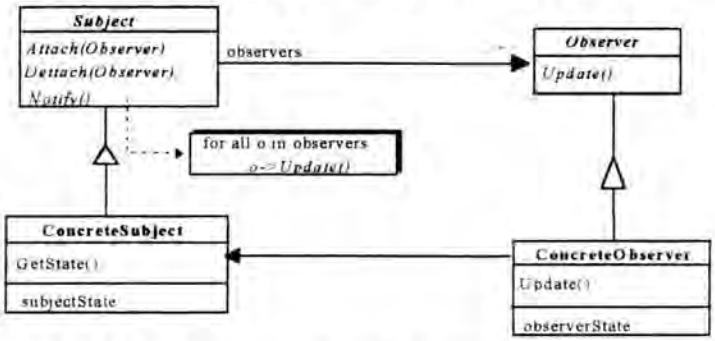


FIGURA A2.21- Estrutura abstrata de classes do padrão *Observer*

A2.2.18 State (Comportamental)

O *State* permite a um objeto mudar seu comportamento quando muda seu estado interno, como se na realidade mudasse de classe. Em geral o padrão é aplicável quando a resposta de um objeto ás mesmas mensagem varia em função de seu estado interno. Cada possível estado é encapsulado por uma classe (*State*) que define o comportamento para esse estado. O objeto original (*StateHolder*) deve criar objetos do tipo correspondente ao seu novo estado e delegar as mensagens ao objeto estado. A separação de estados em classes simplifica o código da classe principal, evitando a codificação de condicionais, embora incremente o número de classes do sistema e, por tanto, a complexidade global.

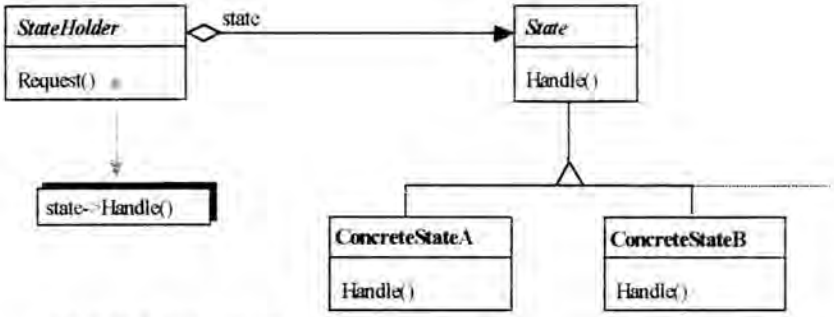


FIGURA A2.22- Estrutura abstrata de classes do padrão *State*

A2.2.19 Strategy (Comportamental)

O objetivo de *Strategy* é encapsular uma família de algoritmos ou comportamentos, de forma tal que estes algoritmos possam ser variados independentemente dos clientes que os usam. O padrão é útil em situações nas quais existem diferentes algoritmos que resolvem o mesmo problema com diferentes restrições de espaço-tempo por exemplo, ou um conjunto de classes relacionadas só se diferenciam numa porção do seu comportamento.

O mecanismo clássico utilizado para obter esta funcionalidade é a herança, no qual subclasses definem as variações de comportamento. No entanto, este esquema fixa a implementação a uma classe específica misturando dados e algoritmos, sem permitir mudar dinamicamente o comportamento. Utilizando o padrão é obter esta funcionalidade, além da possibilidade de compartilhar algoritmos entre diferentes classes.

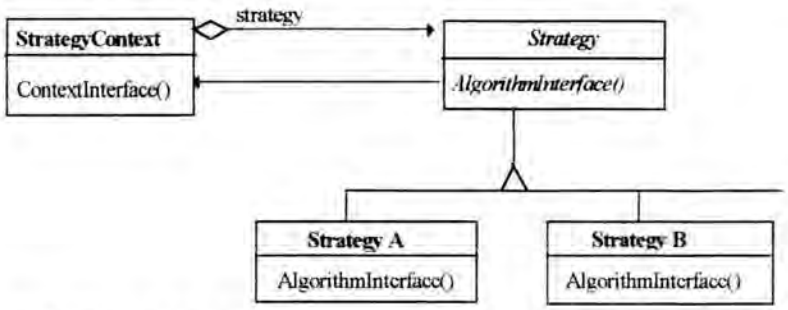


FIGURA A2.23- Estrutura abstrata de classes do padrão *Strategy*

A2.2.20 Visitor (Comportamental)

O *Visitor* encapsula operações a serem realizadas sobre os elementos de uma estrutura heterogênea de objetos, evitando assim acrescentar operações dependentes da aplicação a classes base reutilizáveis. O padrão define duas hierarquias de classes: uma para os elementos sendo operados e outra (os visitadores) para as operações que serão aplicadas sobre os elementos da primeira hierarquia.

O padrão é aplicável em casos em que uma estrutura é composta por objetos de diversas classes e é necessário aplicar uma operação cujo resultado depende da classe de cada objeto na estrutura. Por exemplo, operações a serem realizadas sobre uma árvore de *parsing* por um compilador.

Cada classe de *visitor* definirá um método para tratar cada tipo de objeto, enquanto os objetos a ser visitados deverão informar ao visitor sua identidade através destes métodos. No caso da FIGURA A2.24, os elementos definem o método *Accept* que recebe a instância de *visitor* como argumento. Cada classe responde à mensagem invocando o método do visitor que a identifica, ou seja, *VisitElementA* ou *VisitElementB*. O polimorfismo permite definir genericamente a interface dos *Visitors* através de uma classe abstrata, da qual herdam cada tipo de *Visitor*.

Como é evidente, a interface dos *Visitors* é dependente dos objetos a serem visitados, e se os objetos na estrutura variam freqüentemente, a interface de toda a hierarquia de *visitors* deverá

variar como conseqüência disto. Modificar a interface da hierarquia pode ser uma atividade extremamente trabalhosa, sendo neste caso mais razoável acrescentar as operações nas classes dos elementos da estrutura.

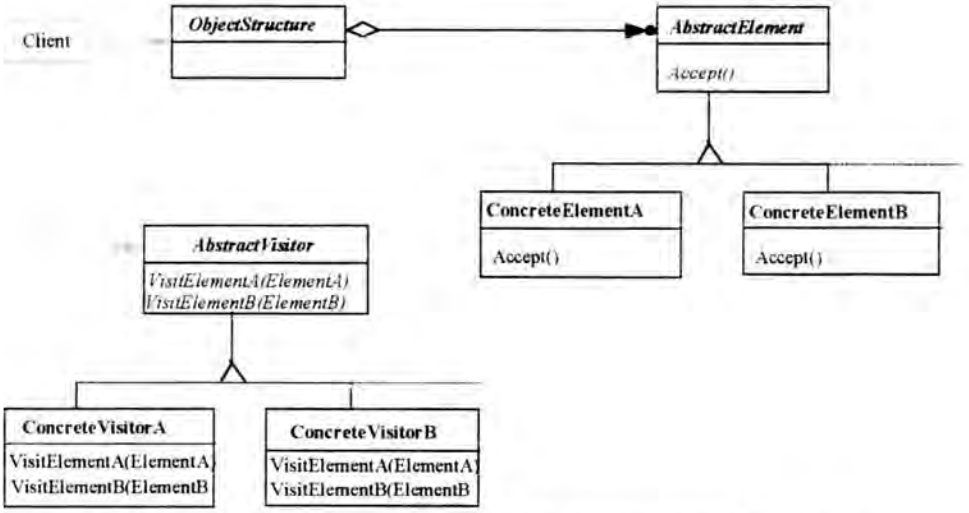


FIGURA A2.24- Estrutura abstrata de classes do padrão *Visitor*

Anexo 3

A Linguagem Smalltalk

Neste anexo apresenta-se uma sinopses da linguagem Smalltalk, com o objetivo de permitir compreender os exemplos de código da tese, àqueles leitores que não conhecem a linguagem. A descrição detalhada pode ser encontrar em [GOL 83].

1.1 Características Gerais

Smalltalk é uma linguagem orientada a objetos pura, não tipada, com herança simples, e encapsulamento estrito. Todas as classes possuem uma superclasse. Object e a raiz da hierarquia completa de classes, definindo o comportamento comum de qualquer objeto no sistema.

Uma classe Smalltalk é um objeto que define a estrutura dos objetos que são suas instâncias. Uma classe descreve o comportamento das instâncias através de *métodos de instância*. O comportamento da classe é definido por *métodos de classe*. Este comportamento consiste, essencialmente, de métodos para criação de instâncias, embora não está restrito só a isso.

Um objeto é identificado por uma referência interna (não acessivel) e tem um estado interno próprio, determinado pelas *variáveis de instância* definidas na sua classe. Adicionalmente, uma classe pode definir *variáveis de classe*, as quais são acessíveis para todas as instâncias dessa classe. As variáveis de classe começam com letra maiúscula.

Uma subclasse define-se com a sintaxe seguinte:

```
<nome superclasse> subclass: #<nome de classe>
    instanceVariables:'<lista de variáveis>'
    classVariables: '<lista de variáveis>'
        .....
```

1.2 Mensagens e Métodos

Os métodos são procedimentos que contém uma seqüência de sentenças terminadas com ponto, as quais implementam um dado comportamento. O retorno de um método é indicado pelo símbolo ^.

Os objetos só se comunicam através da troca de mensagens. Uma mensagem é constituída por um seletor e os argumentos. O seletor define qual método do receptor da mensagem será ativado. A sintaxe de uma mensagem é:

```
receptor mensagem [mensagem...]
```

Uma mensagem pode ter zero ou mais argumentos, os quais são identificados pelo símbolo **:** que separa o nome do método. Isto é, os argumentos são intercalados com o nome do método da forma:

```
r at: 1 put: 2.
```

Neste exemplo o nome do método é ***at:put:*** incluindo os **:** . Os argumentos são 1 e 2. O método correspondente será definido como:

```
at: anIndex put: aValue
        <sentenças>
```

Os nomes dos argumentos são anIndex e aValue, os quais são substituidos por seus valores quando a mensagem é ativada.

Várias mensagens podem ser enviadas em seqüência dentro da mesma sentença. Cada mensagem dentro da seqüência é enviada ao resultado da mensagem precedente. Se as mensagem na seqüência estão separadas pelo símbolo ';', então essas mensagens são enviadas ao objeto receptor da sentença.

A palavra reservada ***self*** referencia ao objeto que recebeu uma dada mensagem. A palavra reservada ***super*** referencia a implementação de um dado método nas superclasses da classe na qual é referenciado. Isto permite executar um método que foi redefinido em uma classe, mas que precisa da implementação herdada. Por exemplo:

self new	produzirá a ativação do primeiro método *new* encontrado na cadeia de herança, **incluindo** a classe na qual é chamado
super new	produzirá a ativação do primeiro método *new* encontrado na cadeia de herança, **excluindo** a classe na qual é chamado.

Um exemplo típico de uso é a redefinição do método de criação de instâncias para acrescentar uma mensagem de inicialização à instância criada:

```
new                                  Neste caso não é possível usar self pois implicaria
    super new initalize              em um ciclo infinito
```

1.3 Blocos

Os blocos são umas das características principais de Smalltalk. Um bloco é um conjunto de sentenças, parametrizado por um conjunto de zero ou mais argumentos, cuja execução é realizada quando o bloco recebe uma mensagem *value*. A sintaxe de um bloco é a seguinte:

```
[ <argumentos> | <sentenças> ]
```

Cada argumento deve ser precedido por um símbolo ':'. Por exemplo,

```
[:v| Transcript show: v printString] value: 1
```

produzirá a impressão da cadeia que representa o valor na console (Transcript) do ambiente (o número 1 neste caso).

1.4 Estruturas de Controle

As estruturas de controle condicional da linguagem são implementadas como mensagens a objetos da classe *Boolean*, que recebem blocos como argumentos, a serem executados de acordo com o valor da condição. Os operadores lógicos também são implementados como mensagens entre objetos da classe *Boolean*.

As estruturas de controle condicionais (*if...*) são implementadas pelas mensagens *ifTrue:*, *ifFalse:*, *ifTrue:ifFalse:*

```
aBoolean
    ifTrue:[<sentenças> ]
    ifFalse:[<sentenças> ].
```

As estruturas de repetição são implementadas como mensagens a um bloco cujo resultado é um *Boolean*.

```
[aBoolean]
    whileTrue:[ <sentenças> ].
```

```
[aBoolean]
    whileFalse:[ <sentenças> ].
```

1.5 Coleções

As coleções de objetos são o principal mecanismo de estruturação de dados fornecido pela biblioteca do ambiente. A classe *Collection* implementa o comportamento geral de coleções de objetos. Subclasses desta classe implementam organizações especificas de objetos, como por exemplo, *OrderedCollection*, *SortedCollection*, *Array*, *Dictionary*, *String*, etc.

As coleções são acessadas pelas mensagens *at:* e *at:put:*. Cada tipo de coleção define o tipo de seus argumentos, isto é, um *Array* só aceita índices numéricos (argumentos da mensagem *at:*), já um *Dictionary* aceita qualquer objeto como chave do dicionário.

As funções de enumeração são os comportamentos mais importantes fornecidos pelas coleções. Por exemplo,

aCollection	**do:**[:v| <sentenças>]	aplica o bloco dado como argumento a cada elemento da coleção
aCollection	**select:**[:v| <condição>]	seleciona os elementos da coleção que tornam verdadeira a condição
aCollection	**detect:**[:v| <condição>] **ifNone:**[<sentenças>]	seleciona o primeiro elemento da coleção que tornam verdadeira a condição. Se não existe nenhum, então executa o bloco, opcional, *ifNone*.

Dicionários:

aCollection	**at:** anObject **ifAbsent:**[<sentenças>]]	retorna o objeto associado com a chave *anObject* se não existe nenhum, executa o bloco, opcional, *ifAbsent*.

Bibliografia

[ABO 95] ABOWD, G.; ALLEN, R.; GARLAN, D. Formalizing Style to Understand Descriptions of Software Architecture. **ACM Transactions on Software Engineering and Methodology**, New York, v.4., n.4, p.319-364, Oct. 1995.

[ADE 85] ADELSON, B.; SOLLOWAY, E. The Role of Domain Experience in Software Design. **IEEE Transactions on Software Engineering**, New York, v. 11, n. 11, p. 1351-1360, Nov. 1985.

[ALE 87] ALEXANDER, J. Paneless panes for Smalltalk Windows. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, LANGUAGES AND APPLICATIONS, 2., 1987. **Proceedings...** New York: ACM Press, 1987.

[AMA 96] AMANDI, A.; PRICE, A. A Linguagem OWB: Combinando Objetos e Lógica. In: SIMPOSIO BRASILERO DE LINGUAGENS DE PROGRAMAÇÃO, 1., 1996, Belo Horizonte. **Proceedings...** Belo Horizonte: SBC, 1996.

[AMA 97] AMANDI, A; PRICE, A. Towards Object-Oriented Agent Programming: The Brainstorm Meta-Level Architecture. In: AUTONOMOUS AGENTS CONFERENCE, 1., 1997, Los Angeles. **Proceedings...**New York:ACM Press, 1997.

[AND 97] ANDRIENKO, G.; ANDRIENKO, N. IRIS: a Knowledge-Based System For Visual Data Exploration. In: CONFERENCE ON COMPUTER-HUMAN INTERACTION, 7., 1997. **Proceedings...** New York :ACM Press, 1997.

[ARA 93] ARANGO, G. et al. A process for Consolidating and Reusing Design Knowledge. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 13., 1993, Baltimore. **Proceedings...** California:IEEE Press, 1993.

[ARA 93a] ARANGO, G. et. al. The Graft-Host Method for Design Change. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 15., 1993, Baltimore. **Proceedings...** Los Alamitos:IEEE Press,1993.

[ARA 91] ARANGO, G; PRIETO DIAZ, R. Domain Analysis Concepts and Research Directions. In: **Domain Analysis Part 1: Introduction and Overview**. Los Alamitos:IEEE Press, 1991.

[ARA 94] ARANGO, G. Domain Analysis Methods. In: SCHÄFER W.; PRIETO-DIAZ, R. MATSUMOTO M. (Eds.) **Software Reusability**. England: Ellis Horwood, 1994. p. 17-47.

[BAT 91] BATORY, D. DaTE: a graphical layout editor for Genesis. In: **The Genesis papers.** Texas, University of Texas, 1991. (Technical Report TR-91-20).

[BAT 92] BATORY,D.; O'MALLEY, S. The Design and Implementation of Hierarchical Software Systems with Reusable Components. **ACM Tansactions on Software Engineering and Methodology**, New York ,v.2 , n.10, Oct. 1992.

[BAT 92a] BATORY, D. et al. Implementing a Domain Model for Data Structures. **International Journal of Software Engineering and Knowledge Engineering**, New York, v.2, n.3, p. 375-402, Sept. 1992.

[BAT 92] BATINI,C.; CERI,C.; NAVATHE,S. **Conceptual Database Design**: An Entity-Relationship Approach. Redwood City: Benjaming-Cummings, 1992. 486p.

[BAX 92] BAXTER, I. Design Maintenance Systems. **Communications of the ACM**, New York, v. 35, n. 4. Apr. 1992

[BEC 94] BECK, K.; JOHNSON, R. Patterns Generate Architectures. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 10., 1994, Bologna, Italia. **Proceedings...** Berlin:Springer-Verlag, 1994. p. 89-110.

[BEG 88] BEGEMAN, M.; CONKLIN, J. The Right Tool for the Job. **Byte**, Peterborough, N. H., v.13., n.10, p. 255-266, Oct. 1988.

[BIG 87] BIGGERSTAFF, T.; RICHTER C. Reusability Framework, Assessment, and Directions. **IEEE Software**, Los Alamitos, v.4 , n. 3, Mar 1987.

[BIG 93] BIGGERSTAF, T.; BHARAT, G.; WEBSTER D. The Concept Assignment Problem in Program Understanding. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 15., 1993, Baltimore. **Proceedings...** Los Alamitos: IEEE Press, 1993.

[BIL 93] BILIRIS, A.; DAR, S.; GEHANI, N. Making C++ Objects Persistent: the hidden pointers. **Software Practice and Experience**, Sussex, England, v.23, n.12, Dec. 1993. p 1285-1303.

[BOO 94] BOOCH, G. **Object-Oriented Analysis and Design**. California:Benjamin/Cummings, 1994. 589 p.

[BRO 83] BROOKS, R. Towards a Theory of the Comprehension of Computer Programs. **International Journal of Man-Machine Studies,** [S.l.], v.18, n.6, p. 543-554, June 1983.

[BRO 84] BROWN, M. Sedgewick, Robert. A system for Algorithm Animation. **ACM Computer Graphics**, New York, v.11, n 7, p. 177-186, July 1984.

[BRO 88] BROWN, M. Exploring Algorithms using Balsa-II. **IEEE Computer**, New York, v.19, n.5, p 14-36, May 1988.

[BRO 93] BROWN, M.; NAJORK, M. **Algoritm Animation Using 3D Interactive Graphics.** Palo Alto, California:Digital Systems Research Center,1993. (Technical Report 110a).

[BRU 93] BRUEGGE, B.; GOTTSCHALK, T.; LUO, B. A Framework for Dynamic Program Analyzers. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, LANGUAGES AND APPLICATIONS, 8., 1993, Washington D.C. **Proceedings...** New York:ACM Press, Oct. 1993.

[BUH 92] BUHR, R.; CASSELMAN, R. Architectures with Pictures. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, LANGUAGES AND APPLICATIONS, 7., 1992, Vancouver, Canadá. **Proceedings...** New York:ACM Press, Oct.1992.

[BUS 96] BUSCHMANN, F. et al. **Pattern - Oriented S. Architecture - A system of patterns**. Baffins Lane:John Wiley & Sons, 1996.

[CAM 95] CAMPO,M.; PRICE, R.T. Meta-Object Support for Framework Understanding Tools. In: ECOOP'95 WORKSHOP ON ADVANCES IN REFLECTION AND META-OBJECT PROTOCOL, 2., 1995, Aarus, Denmark. **Proceedings...**[S.l.: s. n.], 1995.

[CAM 95a] CAMPO, M.; PRICE, R.T. O Uso de Técnicas Visuais e Navegacionais para Compreensão de Frameworks Orientados a Objetos. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 9., 1995, Recife, Brasil. **Anais...** Recife:SBC, 1995. p. 134-148

[CAM 96] CAMPO, M.; PRICE, R. A Visual Reflective Tool for Framework Understanding. In: TECHNOLOGY OF OBJECT-ORIENTED LANGUAGES AND SYSTEMS, 17., 1996, Paris, França. **Proceedings...** Londres:Prentice-Hall, Feb. 1996.

[CAM 96a] CAMPO, M.; PRICE, R. Um Framework Reflexivo para Ferramentas de Visualização de Software. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 10., 1995, São Carlos, Brasil. **Anais...** São Carlos:SBC, 1996. p. 153-169.

[CAM 96b] CAMPO, M.; PRICE, R.T. Meta-Object Manager: A framework for Customizable Meta-Object Support for Smalltalk 80. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, 1., 1996, Belo Horizonte, MG. **Anais...** Belo Horizonte:SBC, 1996.

[CAR 85] CARDELLI, L; WEGNER, P. On Understanding Data Types, Abstraction and Polimorphism. **ACM Computing Surveys**, New York, v.17. , n..4, p. 471-521, Dec. 1985.

[CAS 91] CASANOVA, M. et al. The Nested Context Model for Hyperdocuments. In: HYPERTEXT, 1991, Texas. **Proceedings...** New York: ACM Press, 1991.

[CAS 92] CASSELMAN, R. **A Role-Based Model for Object-Oriented Design**. Vancouver, Canada:Carleton University, 1992. Master Thesis.

[CHA 88] CHANG, S. Visual Languages: A Tutorial and Survey. **IEEE Software**, New York, v.4, n.1, Mar. 1988.

[CHE 88] CHEN, M.; MOUNTFORD, S.; SELLEN, A. A study on interactive 3-D rotation using 2-D control devices. **Computer Graphics**, New York, v.22, n.4, p 121-129, Apr. 1988.

[CHK 90] CHIKOFSKY, E.; CROSS, J. Reverse Engineering and Design Recovery: ATaxonomy. **IEEE Software**, New York, v.7, n.1, p. 13-17, Jan. 1990.

[CHI 93] CHIBA, S.; MASUDA, T. Designing an extensible distributed language with meta-level architecture. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 9., 1993, Kaiserslautern, Germany. **Proceedings...**Berlin: Springer-Verlag, 1993. p. 482-501.

[CHI 95] CHIBA, S. A Metaobject Protocol for C^{++}. **SIGPLAN Notices**, New York, v.30, n.10, p. 482-501, Oct. 1995.

[COA 90] COAD, P.; YOURDON, E. **OOA: Object-Oriented Analysis**. Englewood Cliffs:Prentice Hall, 1990.

[COI 87] COINTE, P. Metaclasses are First Class: the ObjVlisp Model. In: CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS, 2., 1987. **Proceedings...** New York:ACM Press, 1987.

[COL 92] COLEMAN, D.; HAYES, F.; BEAR, S. Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design. **IEEE Transactions on Software Engineering**, New York, v.18, n.1, p. 9-18, Jan. 1992.

[CON 87] CONKLIN, J. Hypertext: An Introduction and Survey. **IEEE Computer**, New York, v. 20, n. 9, p.17-41, Sept. 1987.

[CON 91] CONKLIN, E.; AKEMOVIC, K. A process-Oriented approach to design rationale. **Human-Computer Interaction**, New York, v.6, n. 6, June 1991.

[COP 92] COPLIEN, J. **Advanced C++ Programming Styles and Idioms**. Reading: Addison-Wesley, 1992.

[CUR 89] CURTIS, B. Cognitive Issues in Reusing Software Artifacts. In: BIGGERSTAF, T., PERLIS, A. (Eds.) **Software Reusability**: Applications and Experience. New York:ACM Press, 1989.

[DEP 93] DE PAUW, W. et al. Visualizing the Behavior of Object-Oriented Programms. **SIGPLAN Notices**, New York ,v.28, n.10, p.326-337, Oct.1993.

[DEP 94] DE PAUW, W. et al. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 10., 1994, Bologna, Italia. **Proceedings...** Berlin:Springer-Verlag, 1994. p. 175-194.

[DEU 89] DEUTSCH, P. Frameworks and reuse in the Smalltalk-80 system. In: BIGGERSTAF, T., PERLIS, A. (Eds.) **Software Reusability**: Applications and Experience. New York: ACM Press, 1989.

[FER89] FERBER, J. Computational Reflection in Class Based Object-Oriented Languages. **SIGPLAN Notices**, New York, v.24, n.10, p. 317-326, Oct. 1989.

[FIS 91] FISHER, G.; HENNINGER, S.; REDMILES, D. Cognitive Tools for Locating and Comprehending Software Objects for Reuse. In: CONFERENCE ON SOFTWARE ENGINEERING, 13., 1991. **Proceedings...** Los Alamitos: IEEE Press, 1991, p. 318-328.

[FIS 95] FISHKIN, K.; STONE, M. Enhanced Dynamic Queries via Movable Filters. In: CONFERENCE ON COMPUTER-HUMAN INTERACTION, 1995. **Proceedings**... New York:ACM Press, p. 415-420,1995.

[GAB 91] GABRIEL, P.; WHITE, J.; BOBROW, P. CLOS: Integrating Object-Oriented and Functional Programming. **Communications of the ACM**, New York, v.34, n.9, Sept. 1991.

[GAM 92] GAMMA, E. ET++SwapsManager: Using Object Technology in the Financial Engineering Domain. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, LANGUAGES AND APPLICATIONS, 7., 1992, Vancouver, Canadá. **Proceedings...** New York:ACM Press, 1992.

[GAM 93] GAMMA, E. et al. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 9., 1993, Kaiserslautern, Germany. **Proceedings...** Berlin:Springer-Verlag, 1993.

[GAM 94] GAMMA, E. et al. **Design Patterns:** Reusable Elements of Object-Oriented Design. Reading: Addison-Wesley, 1994.

[GAR 94] GARLAN, D.; SHAW, M. An Introduction to Software Architecture. In: AMBRIOLA V.; TORTORA G.(Eds.). **Advances in Software Engineering and Knowledge Engineering**, Mass:Carnegy-Mellon University, 1994. Technical Report CMU-CS-94-166.

[GOL 83] GOLDBERG, A. **Smalltalk-80:** The Language and its Implementation. Reading:Addison-Wesley, 1983.

[GOL 95] GOLDBERG, A.; RUBIN, K. **Succeeding with Objects:** Decision Frameworks for Projet Management. Reading: Addison-Wesley, 1995.

[GRA 89] GRAUBE, N. Metaclass compatibility. **SIGPLAN Notices**, New York, v.24, n.10, p.305-315, Oct. 1989.

[GUT 90] GUTFREUND, S. Manipulating Icons in ThinkerToy. In: GLINERT, E (Ed.). **Visual Programming Environments:** Applications and Issues, California:IEEE Press, 1990.

[HEL 90] HELM, R. et al. Contracts: Specifying Behavioral Compositions in Object Oriented Systems. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, LANGUAGES AND APPLICATIONS, 5., 1990, Otawa, Canada. **Proceedings...** New York: ACM Press, 1990.

[HIN 94] HINKLE, B. Reflection in the Smalltalk-80 System. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, LANGUAGES AND APPLICATIONS, 8., 1994. **Tutorial Notes**... [S.l:s.n], 1994.

[IBM 93] IBM CORPORATION. **VisualAge 1.0 Refence Manual**, [S.l.]:IBM, 1993.

[IRA 94] IRASSAR, P.; CAMPO, M.; AMANDI, A. **Redes Hipermedia como soporte para Ambientes de Desarrollo de Software**. Tandil: Universidad Nacional del Centro, Fac. Cs. Exactas, 1994. (ISISTAN TR-03-94).

[JAC 92] JACOBSON, I.; CHRISTERON, M.; OVERGAARD, G. **Object-Oriented Software Engineering:** A Use Case Driven Approach. Reading: Addison-Wesley, 1992.

[JER 96] JERDING, D.; STASKO, J.; BALL, T. **Visualizing Message Patterns in Object-Oriented Program Executions**. Georgia:Georgia Technology Institute, 1996. (Tech. Report GIT-GVU-96-15).

[JOH 88] JOHNSON, R.; FOOTE, B. Designing Reusable Classes. **Journal of Object-Oriented Programming**, New York, v.1, n.12, 1988.

[JOH 89] JOHNSON, R.; FOOTE, B. Reflective Facilities in Smalltalk-80. **SIGPLAN Notices**, New York, v.24, n.10, Oct. 1989.

[JOH 92] JOHNSON, R. Documenting Frameworks Using Patterns. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, LANGUAGES AND APPLICATIONS, 7., 1992, Vancouver, Canadá. **Proceedings...** New York:ACM Press, 1992.

[JOH 93] JOHNSON, R. How to Design Frameworks. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, LANGUAGES AND APPLICATIONS, 8., 1993, Washington DC. **Tutorial Notes**... [S.l:s.n], 1993.

[KIC 91] KICZALES G.; DES RIVIERES, J.; BOBROW; D. **The Design and Implementation of Meta-Object Protocols**.Cambridge: MIT Press, 1991. 335 p.

[KIC 92] KICZALES, G. Towards a New Model of Abstraction in Software Engineering. In: INTERNATIONAL WORKSHOP ON NEW MODELS FOR SOFTWARE ARCHITECTURE/ REFLECTION AND META-LEVEL ARCHITECTURES, 1992,Tokyo, Japan. **Proceedings**... [S.l.:s.n.], 1992, p.1-11.

[KNU 84] KNUTH, D. Literate Programming. **Computer Journal**, New York, v.27, p. 97-111. May 1984.

[KOI 93] KOIKE, H. The Role of Another Spatial Dimension in Software Visualization. **ACM Transactions on Information Systems**, New York, v.11, n.3, p. 266-286, July 1993.

[KRU 92] KRUEGER , C. Software Reuse. **ACM Computing Surveys**, New York, v.24, n.2, p. 132-182, June 1992.

[LAJ 94] LAJOIE, R.; KELLER, R. Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert. In: CONGRESS OF THE ASSOCIATION CANADIENNE-FRANÇAISE POUR L' AVANCEMENT DES SCIENCES, 62., 1994, Montreal, Canada. **Proceedings...** [S.l: s.n], 1994.

[LAN 89] LANGE, B; MOHER, T. Some Strategies of Reuse in an Object-Oriented Programming Environment. In: HUMAN FACTORS IN COMPUTING SYSTEMS, 1989, Austin, Texas. **Proceedings...** New York: ACM Press, 1989. p. 69-73

[LAN 95] LANGE, D.; NAKAMURA Y. Interactive Visualization of Design Patterns Can Help in Framework Understanding. **SIGPLAN Notices**, New York, v.30, n.10. Oct. 1995.

[LIS 95] LISBOA, M. L. **MOTF: Meta-Objetos para Tolerância a Falhas**. Porto Alegre: CPGCC da UFRGS, 1995. Tese de doutorado.

[LOR 94] LORENZ, M; KID, J. **Object-Oriented Software Metrics - A practical guide.**. Englewood Cliffs: Prentice-Hall, 1994.

[MAC 91] MACKINLAY, J.; ROBERTSON, G.; CARD, K. The Perspective Wall: Detail and Context Smoothly Integrated. In: CONFERENCE COMPUTER-HUMAN INTERACTION, 1991. **Proceedings...** NewYork: ACM Press, 1991,p. 173-179.

[MAD 95] MADSEN, O. Open Issues in Object-Oriented Programming. **Software Practice and Experience**, Sussex:England, v.25, n.4, p. 3-43, Dec. 1995.

[MAE 87] MAES, P. Concepts and Experiments in Computational Reflection. **SIGPLAN Notices**, New York, v.22 , n.12, p. 147-169.

[MAE 88] MAES, P. Issues in Computational Reflection. In: MAES,P.; NARDI, D. (Eds.). **Meta-Level Architecture and Reflection.** Amsterdam: Elsevier Science, 1988. p. 21-35.

[MAR 95] MARCHIONINI, G. **Information Seeking in Electronic Environments**. Cambridge: Cambridge University Press, 1995.

[MAS 92] MASUHARA, H. et al. Object Oriented Concurrent Reflective Languages can be Implemented Efficiently. In: CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATION, 7., 1992, Vancouver, Canada. **Proceedings...** New York: ACM Press, 1992.

[MAY 95] MAYRHAUSER VON, A.; VANS, A. Program Comprehension During Software Maintenance and Evolution. **IEEE Computer**, New York, v.24, n.8, p. 44-55, Aug. 1995.

[MIL 56] MILLER, G. The Magical Number Seven, Plus or Minus Two: Limits of Our Capacity to Process Information. In: GLINERT, E. (Eds.). **Visual Softwre Development Environments**: Applications and Issues. California:IEEE Press, 1990. p. 276-291.

[MUL 95] MULET, P; MALENFANT, J; COINTE, P. Towards a Methodology for Explicit Composition of MetaObjects. **SIGPLAN Notices**, New York, v.30, n.10, p.316-330, Oct. 1995.

[MUT 95] MUTHUKUMARASAMY, J.; STASKO, J. **Visualizing Program Executions on Large Data Sets using Semantic Zooming**. Georgia:Georgia Institute of Technology, 1995. (Tech. Report. GIT-GVU-95-02).

[MYE 86] MYERS, B. Visual Programming. Programming by example and program visualization: a taxonomy. In: INTERNATIONAL CONFERENCE ON COMPUTER-HUMAN INTERACTION, 3., 1986. **Proceedings...** California: IEEE Press, 1986.

[OPD 92] OPDYKE, W. **Refactoring Object Oriented Frameworks**. Urbana-Champaign: University of Illinois, 1992. Ph.D. Thesis.

[ORO 95] OROSCO, R.; CAMPO, M.; SOLÉ, J. Mirror: Visually Reflecting C++. In: TECHNOLOGY OF OBJECT-ORIENTED LANGUAGES AND SYSTEMS, 15., 1995, Santa Bárbara. **Proceedings**... California:Prentice-Hall, 1995.

[ORT 95] ORTIGOSA, A. **Proposta de um Ambiente Adaptável de Apoio ao Processo de Desenvolvimento de Software**. Porto Alegre: CPGCC da UFRGS, 1995. Dissertação de Mestrado.

[ORT 96] ORTIGOSA, A.; CAMPO, M. Architectural-Driven Analysis of C++ Program Behavior Using Meta-Objects. In: TECHNOLOGY OF OBJECT-ORIENTED LANGUAGES AND SYSTEMS, 18., 1996, Santa Bárbara, California. **Proceedings...** California:Prentice-Hall, 1996.

[PAR 79] PARNAS, D. Designing software for ease extension and contraction. **IEEE Transactions on Software Engineering**, New York, v.5, n.2, p. 128-137, Feb. 1979.

[PAR 94] PARCPLACE INC. **VisualWorks 2.0 Reference Manual**. California:Parcplace, 1994.

[PEN 87] PENNINGTON, N. Comprehension Studies in Programming. In: EMPIRICAL STUDIES OF PROGRAMMERS, 2., 1987. **Proceedings...** Norwood:Ablex Publishing, 1987. p. 100-112.

[PIN 90] PINTADO, X. et al. Class Management for Software Communities. **Communications of the ACM**, New York, v.33, n. 9, p. 90-102, Sept. 1990.

[PRE 94] PREE, W. Meta-Patterns: Abstracting the Essentials of Object-Oriented Frameowrks. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 10., 1994, Bologna, Italia. **Proceedings...** Berlin:Springer-Verlag, 1994. p. 150-164.

[ROB 93] ROBERTSON, G. Information Visualization using 3D Interactive Animation. **Communications of the ACM**, New York, v.36, n. 4, Apr. 1993.

[ROG 93] ROGOWITZ, B.; TREINISH, L. An architecture for rule-based visualization. In: VISUALIZATION, 1993. **Proceedings...** California:IEEE Press, 1993.

[ROM 93] ROMAN, G.; COX, K. Program Visualization: The Art of Mapping Programs to Pictures. In: CONFERENCE ON SOFTWARE ENGENEERING, 14., Melbourne, Australia, 1993. **Proceedings...** California: IEEE Press, 1993. p. 412-420.

[RUM 91] RUMBAUGH, J. et al. **Object-Oriented Modelling and Design**. Uper Saddle River:Prentice-Hall, 1991.

[SCH 87] SCHNEIDEWIN, N. The State of Software Maintenance. **IEEE Transactions on Software Engineering**, New York, v.13, n.3, p. 303-310, Mar 1987.

[SHA 96] SHAW, M.; GARLAN, D. **Software Architecture - Perspectives on an Emerging discipline**. Uper Saddle River: Prentice - Hall, 1996.

[SHL 88] SHLAER, S.; MELLOR, S. **Object-Oriented Systems Analysis**: Modelling the World in Date. Englewood Cliff: Yourdon Press, 1998. 144 p.

[SHN 87] SHNEIDERMAN, B. **Designing the User Interface**: Strategies for Effective Human-Computer Interaction. Reading: Aison-Wesley, 1987.

[SIL 95] SILVA, J.; PÁEZ, L.; MARCHIONINI, G. Evaluating User Disorientation: A Comparison of Hypertext and Continuous Zooming Interfaces. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 10., 1995, São Carlos, Brasil. **Anais...** São Carlos: SBC. 1996.

[SMI 87] SMITH, R. Experience with the Alternate Reality Kit: An Example of the Tension between Literalism and Magic. In: INTERNATIONAL CONFERENCE on HUMAN-COMPUTER INTERACTION, 1987. **Proceeedings...** New York: ACM Press, 1987.

[SOL 84] SOLLOWAY, E; EHRLICH, K. Empirical Studies of Programming Knowledge. **IEEE Transactions on Software Engineering**, New York, v.10, n.5, p. 595-609, May 1984.

[STA 90] STASKO, J. Simplifying Algorithm Animmation with TANGO. In: IEEE WORKSHOP ON VISUAL LANGUAGES, 1990, Stockie, Illinois. **Proceeedings...** Los Alamitos:IEEE Press, 1990.

[STA 94] STASKO, J.; JERDING, D. **Using Visualization to Foster Object-Oriented Program Understanding**. Atlanta:Georgia Institute of Technology, 1994. (Technical Report GIT-GVU-94-33).

[STE 94] STEEL, L. Beyond objects. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 8., 1994, Bologna, Italy. **Proceedings**... Berlin: Springer-Verlag, 1994. p.1-11.

[STR 91] STROUSTRUP, B. **The C^{++} Programming Language**. 2. ed. Reading: Addison Wesley, 1991.

[SZE 88] SZEKELY, P **Separating User Interface from the Functionality of Application Programs.** Mass:Carnegie-Mellon University, 1988. PhD. Thesis.

[TAE 89] TAENZER, D.; GANTI, M.; PODAR, S. Object-Oriented Software Reuse: The Yo-Yo Problem. **Journal of Object-Oriented Programming**, New York, v.2, Sept. 1989.

[TIL 96] TILEY, S; SANTANA, P. Towards a Framework for Program Understanding. In: IEEE INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION, 4., 1996, Berlin, Alemanha. **Proceedings...** Los Alamitos: IEEE Press, 1996.

[VIO 94] VION-DURY, J.; SANTANA, M. Virtual Images: Interactive Visualization of Distributed Object-Oriented Systems. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, LANGUAGES AND APPLICATIONS, 9., 1994, Portland, Oregon. **Proceedings...** New York: ACM, 1994.

[VLI 89] VLISSIDES J.; LINTON M. Composing User Interfaces with InterViews. **IEEE Computer**, New York,v.22, n.2, Feb. 1989.

[WAT 88] WATANABE, T.; YONEZAWA, A. Reflection in an Object-Oriented Language. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, LANGUAGES AND APPLICATIONS, 3., 1988, San Diego, California. **Proceedings...** New York:ACM Press, 1988.

[WEI 84] WEISER, M. Program Slicing. **IEEE Transactions on Software Engineering**, New York, v.10, n.4, p.352-357, July 1984.

[WIL 89] WILDE, N.; HUITT, R. Dependency Analysis Tools: Reusable Components for Software Maintenance. In: IEEE CONFERENCE ON SOFTWARE MAINTENANCE, 1., 1989. **Proceedings...**[S.l.:s.n], 1989.

[WIL 92] WILDE, N.; HUIT, R. Maintenance Support for Object-Oriented Programs. **IEEE Transactions on Software Engineering**, New York, v.18, n.12, Dec.1992.

[WIN 83] WINSTON, P. **Artificial Intelligence.** Reading:Addison-Weslley, 1983.

[WIT 96] WINOGRAD, T. **Bringing Design to Software**. New York: ACM Press, 1996.

[WIR 90] WIRFS-BROOKS R.; JOHNSON R. Surveying Current Research in Object Design. **Communications of the ACM**, New York, v.33, n.9, Sept.1990.

Informática UFRGS

CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Compreensão Visual de Frameworks Através da Introspeção de Exemplos

por

Marcelo Ricardo Campo

Defesa de Tese apresentada aos Senhores:

Prof. Dr. Alain Pirotte (Louvain la Neuve)

Prof. Dr. Júlio César Sampaio do Prado Leite (PUC/RJ)

Prof. Dr. Carlos Alberto Heuser

Vista e permitida a impressão.
Porto Alegre, 18/06/97.

Prof. Dr. Roberto Tom Price,
Orientador.

Prof. Flávio Rech Wagner
Coordenador do Curso de Pós-Graduação
em Ciência da Computação - CPGCC
Instituto de Informática - UFRGS