Um Framework Reflexivo para Ferramentas de Visualização de Software

Marcelo R. Campo* - Roberto Tom Price

Universidade Federal do Rio Grande do Sul - Instituto de Informática
Caixa Postal 15064 - Porto Alegre - RS- Brasil

* também Universidad Nacional del Centro de la Provincia de Bs. As.
Fac. de Ciencias Exactas - ISISTAN
San Martín 57, (7000) Tandil, Bs. As., Argentina

email: {mcampo,tomprice }@inf.ufrgs.br

Abstract

This paper presents Luthier, a Smalltalk-80 framework, designed to support the construction of visual tools for dynamic program analysis. Luthier integrates computational reflection techniques based on meta-objets with hypertext and direct-manipulation user interfaces techniques, especially designed to build dynamically adaptable visualization tools through composition of basic behaviours. The framework introduces two main contributions: a meta-object protocol based on the concept of *meta-objet managers* - which allow the construction of specialized mechanisms for meta-object association and activation - and the explicit support for *abstraction scales* for the dynamic (and continuos) reconfiguration of visualizations allowing different levels of abstraction. Through the combination of these mechanisms it is possible to build highly-complex structures independently of the implementation of specific visualizations. This allows a greater level of reusability of visualizations, which it is often the most limiting aspect of software visualization systems.

Keywords: object-oriented frameworks, software visualization, computational reflection, meta-object managers, software reuse.

1. Introdução

Na medida que as aplicações orientadas a objetos tornam-se mais complexas, ferramentas visuais para suportar as tarefas de desenvolvimento, depuração, manutenção e reutilização tornam-se progressivamente mais necessárias. Os mecanismos de herança, polimorfismo e acoplamento dinâmico permitem tanto a construção de sistemas altamente flexíveis quanto de componentes genéricos e extensíveis em forma consistente dentro do paradigma. Estes mecanismos, porém, fazem os programas orientados a objetos mais difíceis de compreender e manter.

Nos últimos dois anos, os autores, tem estado envolvidos num projeto de pesquisa sobre a utilização de ferramentas visuais para apoio à *compreensão* de software orientado a objetos, com ênfase especial no apoio à reutilização de *frameworks* [Campo 95a][Campo 95b][Campo 96b]. Um *framework* [Deutsch 89] é uma infra-estrutura ou esqueleto de uma família de aplicações, projetado para ser reutilizado. 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.

Como resultado dos primeiros desenvolvimentos, uma ferramenta atualmente denominada Luthier, foi desenvolvida em Smalltalk-80. Esta ferramenta se carateriza pela utilização técnicas de análise do comportamento dinâmico de exemplos construídos com um dado framework para gerar uma visualização arquitetônica do framework. Esta visualização é baseada em uma notação especialmente projetada para ressaltar os aspectos essenciais da arquitetura abstrata dos frameworks (classes abstratas, categorias de métodos e fluxo abstrato de controle). A Fig. 1 mostra uma imagem da ferramenta analisando um editor gráfico produzido com o framework HotDraw [Johnson 92]. A visualização é complementada com técnicas de navegação entre a representação gráfica e o código, e animação da execução. Para demonstrar a efetividade dos

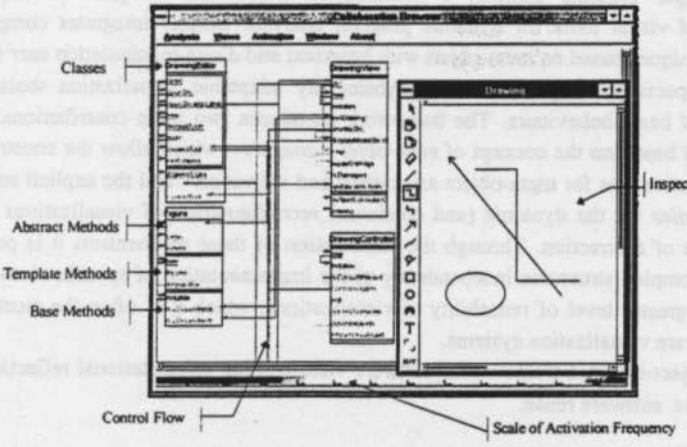

Fig. 1. Imagem da representação estrutural provida por Luthier

mecanismos providos, foram realizados vários experimentos com usuários não experientes na utilização de frameworks, divididos em um grupo utilizando a ferramenta e outro não. O resultado desses experimentos mostrou que aqueles usuários que utilizaram Luthier produziram aplicações de maior qualidade em termos de reutilização. Entretanto, uma limitação essencial da ferramenta também se fez evidente: usuários requerem níveis de abstração diferentes.

As limitações mais relevantes foram as seguintes:

- Uma representação visual centrada nos aspectos estruturais de classes não se mostrou suficiente para aqueles usuários acostumados a pensar em termos de instâncias.
- Novas representações visuais são difíceis de serem compreendidas por usuários acostumados a notações providas pelos métodos de desenvolvimento já estabelecidos.
- O volume de informação que pode ser recuperado para frameworks complexos é muito grande, não sendo suficientes os mecanismos de navegação e animação providos.
- Uma representação estrutural enfatizando troca de mensagens entre classes abstratas é de utilidade para compreender o projeto detalhado do framework, mas dificulta o processo inicial de compreender o framework num nível mais alto de abstração.

Como conclusão destas limitações mostraram-se necessários mecanismos que permitam *adaptar* a ferramenta aos diferentes níveis de abstração nos quais o software pode ser representado, bem como às diferentes formas de representação preferidas dos usuários.

Estes requisitos determinaram a evolução do suporte de Luthier para um *framework* para construção de ferramentas de visualização, que permita um alto grau de flexibilidade para a adaptação do comportamento das ferramentas em tempo de execução, tanto nos aspectos de captura de informação quanto nos aspectos de abstração e técnicas de visualização. A descrição dos mecanismos mais relevantes deste framework é o objetivo deste artigo.

A seção seguinte apresenta as dimensões essenciais de projeto que definem as principais decisões de projeto de Luthier. Na seção 3 apresenta-se a descrição geral do framework. A seção 4 discute o papel do padrões de projeto no projeto do framework. Na seção 5 descrevem-se os trabalhos relacionados mais relevantes na área e finalmente, na seção 6, discutem-se linhas futuras de evolução do framework.

2. Dimensões de Projeto

O conceito de *framework* está relacionado com o conceito de *modelo de um domínio de aplicação*. Basicamente, um modelo de domínio identifica as entidades genéricas que compõem o domínio e os relacionamentos entre elas [Arango 94]. Um *framework* representa uma implementação orientada a objetos desse modelo em termos de classes abstratas, definindo as interfaces entre elas em termos de métodos que implementam os mecanismos genéricos que caraterizam o *comportamento* das aplicações do domínio. Assim, uns dos problemas centrais

detrás do projeto do um *framework* reside na identificação do conjunto não redundante de abstrações de comportamento que caraterizam o domínio.

Expressado em outros termos, o projeto de um framework requer a identificação adequada das dimensões essenciais através das quais o domínio de aplicação pode ser caraterizado. No domínio dos sistemas de visualização de software, Roman e Cox [Roman 93] definem um espaço de quatro dimensões que os carateriza: escopo, abstração, técnica e método de especificação.

- ***Escopo***: representa aqueles aspectos do programa a serem a visualizados como código, dados, estados ou comportamento. Esta dimensão identifica a natureza da informação que deverá ser obtida para ser visualizada. Em geral os sistemas de visualização enfatizam alguns destes aspectos sobre os outros.
- ***Abstração***: Esta dimensão identifica a natureza do mapeamento de conceitos do programa a representações gráficas, como *representação direta* (ex. visualização de listas encadeadas), *representação estrutural* (ex. visualização de grafos de controle) ou *representação sintetizada* (ex. informação derivada como visualização de subsistemas), entre outras.
- ***Técnica***: refere-se aos aspectos sintáticos e semânticos da representação gráfica e às facilidades de organização e manipulação dessa representação, como por exemplo linguagem visual ou projeto da interface.
- ***Método de Especificação***: define o mecanismo utilizado para *capturar informação* a ser visualizada (ex. anotação de código, predefinição, associação).

Cada uma destas dimensões identifica um conjunto de diferentes alternativas, as quais podem ser implementadas através de um framework. Assim, Luthier foi projetado como a composição de quatro sub-frameworks, os quais são descritos a seguir.

3. O Framework Luthier

A Fig. 2 apresenta a arquitetura global de Luthier, composta por quatro sub-frameworks que fornecem a infra-estrutura extensível para implementar mecanismos especializados para:

- Captura de Informação através de Meta-Objetos
- Representação da Informação Capturada
- Construção de Abstrações
- Construção de Visualizações

Uma ferramenta típica construída com o suporte fornecido por 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 objetos encarregados de selecionar ou construir abstrações da informação contida na representação de hiperdocumento. Nas seções seguintes as caraterísticas mais importantes dos

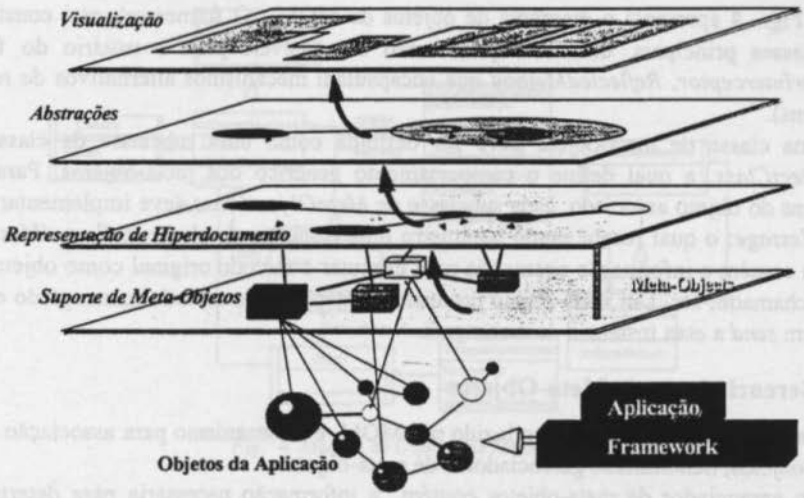

Fig. 2. Arquitetura Global de Luthier

quatro frameworks são descritas, colocando especial ênfase nas contribuições mais relevantes de Luthier: captura de informação e construção de abstrações.

3.1 Captura de Informação: O Sub-framework de Meta-Objetos

Este aspecto representa uns dos fatores chave na facilidade com a qual um usuário pode construir a visualização desejada. As atividades de depuração ou teste requerem um alto grau de flexibilidade para especificar o tipo de mapeamento entre construções do programa e a sua visualização, também como a capacidade de controlar a sua execução.

Para esta tarefa, Luthier, diferente das abordagens habituais baseadas em anuncio de eventos de interesse [Bruegge 93][De Pauw 93][Stasko 94][Lange 95], adota um mecanismo reflexivo baseado em *meta-objetos* [Maes 87]. Sob esta abordagem um objeto a ser analisado possui um meta-objeto associado que intercepta as mensagens para monitorar ou controlar o comportamento desse objeto. Este tipo de meta-objetos oferece um alto nível de flexibilidade para construir bibliotecas de comportamentos reflexivos, os quais podem ser *acrescentados* em forma *dinâmica* e *não intrusiva* (a respeito do código fonte) a um sistema existente para monitorar seu comportamento.

O sub-framework para suporte de meta-objetos, chamado MOM, 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 a um conjunto de instâncias de diferentes classes; a uma classe ou a um conjunto de classes. Um método, instância ou classe pode ter vários meta-objetos associados.

A Fig. 4 apresenta o diagrama de objetos de MOM . O framework está constituído por cinco classes principais, duas das quais não são visíveis para o usuário do framework (*WrapperInterceptor*, *ReflectedMethod* que encapsulam mecanismos alternativos de reflexão de mensagens).

Uma classe de meta-objeto deve ser definida como uma subclasse da classe abstrata *MetaObjectClass*, a qual define o comportamento genérico dos meta-objetos. Para tratar as mensagens do objeto associado, cada subclasse de *MetaObjectClass* deve implementar o método *handleMessage*: o qual recebe como parâmetro uma instância da classe *ReflectedMessage*. Esta instância contém a informação necessária para executar o método original como objeto receptor, método chamado, etc. Um meta-objeto produzirá a execução do método interceptado enviando a mensagem *send* a essa instância de mensagem.

3.1.1 Gerenciadores de Meta-Objetos

O aspecto mais relevante introduzido por MOM, é o mecanismo para associação e ativação de meta-objetos, denominado gerenciadores 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. Quando uma mensagem é capturada, é redigida para um dado gerenciador o qual encapsula o mecanismo que seleciona o meta-objeto que deverá tratar a mensagem enviando para ele a mensagem *handleMessage*. A Fig. 3 apresenta um esquema do mecanismo de reflexão de mensagens baseado em gerenciadores. Quando um objeto refletido recebe uma mensagem, esta mensagem é desviada pelo mecanismo de intercepção para o gerenciador associado. O gerenciador decide quando transferir o controle para um meta-objeto ou executar o método original.

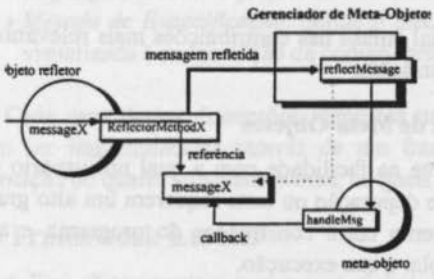

Fig. 3. Mecanismo de Reflexão baseado em Gerenciadores

O mecanismo genérico dos gerenciadores é implementado pela classe *MetaObjectManager*, a qual pode ser especializada para implementar mecanismos alternativos de ativação de meta-objetos.

Para isto uma subclasse de *MetaObjectManager* deverá prover a implementação dos métodos: *findMetaObjectsFor:* e *activateMetaObject:withMessage:* para encontrar o(s) meta-objeto(s) associados com o receptor da mensagem e para transferir o controle para esse(s) meta-objeto(s).

Por exemplo, Luthier utiliza um gerenciador, chamado *LayeredManager*, que associa meta-objetos *default* com classes. Quando uma mensagem é refletida, procura aquele meta-objeto associado com a classe que implementa a mensagem na hierarquia de herança e transfere o controle para esse meta-objeto. Esta funcionalidade é implementada da maneira seguinte:

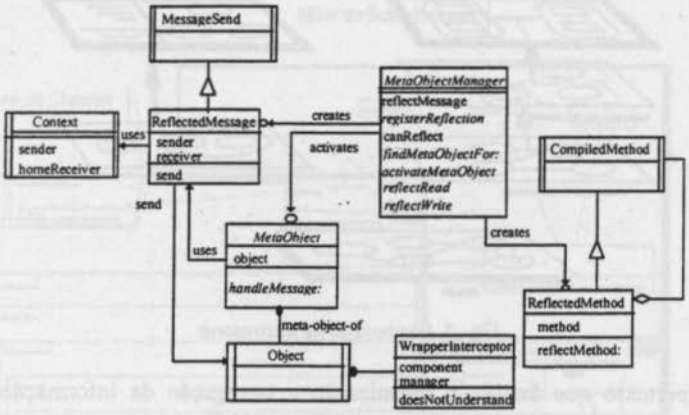

Fig. 4. Modelo de Objetos de MOM

```
findMetaObjectsFor: aReflectedMessage
        |mo|
        mo:= ReflectedClasses at: aReflectedMessage methodImplementorClass.
        ^mo

activateMetaObject: aCollection withMessage: aReflectedMessage
                "Activates all the meta-objects associated"
        aCollection do:[ :mo| mo handleMessage: aReflectedMessage].
                "Returns the return 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.

3.2 O Sub-Framework de Representação

O sub-framework de representação é baseado no modelo PROMETO [Ortigosa 95], especialmente projetado para suportar ambientes de desenvolvimento de software. O modelo se compõe de três classes de componentes: *Entidades* (representam elementos básicos), *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 somente admite como integrantes instâncias das classes que tem sido definidas para sua classe. 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.

Através do suporte provido pelo framework é possível representar a informação do programa analisado de acordo com as necessidades da ferramenta específica, segundo uma

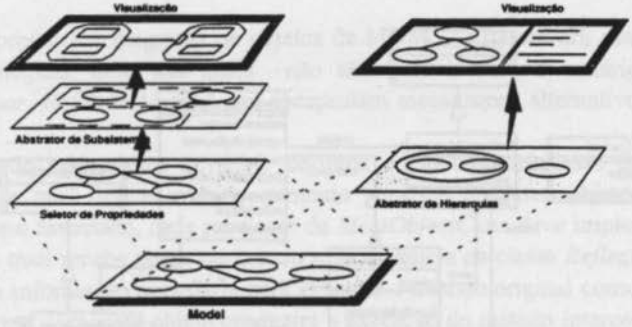

Fig. 5. Composição de Abstratores

estrutura de hipertexto que facilita a organização e navegação da informação. A descrição detalhada deste framework pode-se encontrar em [Ortigosa 95].

3.3 O Sub-Framework de Abstrações

A facilidade para construir abstrações é um dos aspectos centrais em sistemas de visualização de software. Do ponto de vista do domínio dos analisadores de programas, abstrações de software podem ser consideradas como aquelas construções que não são suportadas em forma explícita pela linguagem de programação, mas que formam parte integral da forma na que o software é conceitualizado pelo projetista. Subsistemas [Wirf 90], grupos colaborativos [Helm 90] ou padrões de projeto [Gamma 94] representam abstrações de projeto que não são habitualmente 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.

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 vista 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. Entretanto, a forma habitual para implementar estes mecanismos requer que cada tipo de visualização recupere a informação necessária e produza a representação gráfica de acordo com uma notação determinada. Isto faz com que as visualizações sejam altamente dependentes da representação da informação, como também que mecanismos para filtrar os diferentes níveis de abstração devam ser implementados para cada visualização particular, reduzindo a reutilizabilidade das visualizações.

Para evitar este inconveniente, o sub-framework de abstrações provê um mecanismo genérico para representar abstrações de software: o *abstrator*.

Um abstrator é um objeto intermediário entre os objetos visuais e a representação de informação. Um abstrator pode agir como um *seletor* ou *gerador* da informação que uma

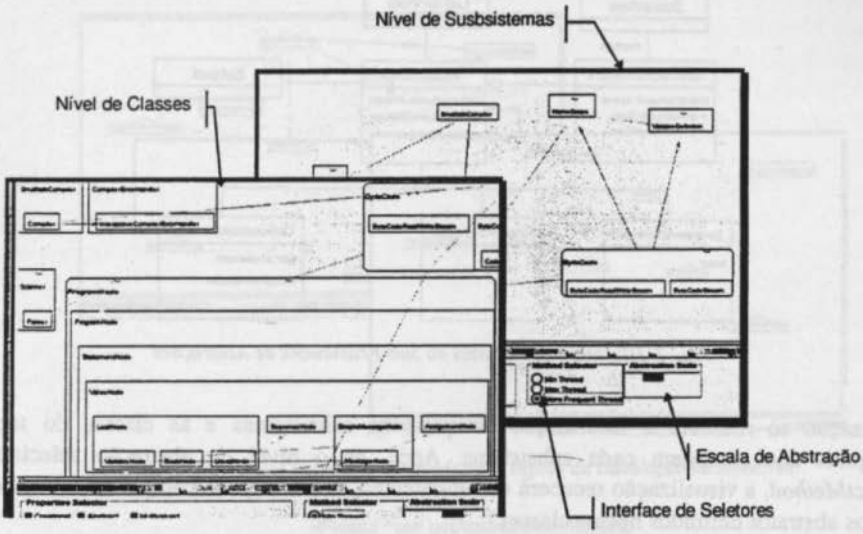

Fig. 6. Exemplo da variação da escala de abstração em uma visualização produzida por Luthier

visualização mostrará graficamente. Por exemplo, um abstrator pode encapsular o algoritmo para reconhecer subsistemas ou grupos colaborativos e prover cada subsistema reconhecido como um dado a ser visualizado. Também pode selecionar, de acordo com algum critério estabelecido interna ou externamente, qual a informação a ser visualizada e ser composto por outros abstratores (Fig. 5) para gerar e selecionar a informação necessária pela visualização.

Através do uso de abstratores as visualizações só devem se preocupar em mostrar graficamente a informação disponível, sem tomar em consideração o grau de detalhe das mesmas. Esta habilidade é de grande importância no caso de abstrações compostas, como por exemplo subsistemas. O grau de detalhe da apresentação pode ser controlado pelos abstratores, permitindo assim transformar gradualmente o nível de detalhe apresentado através da mudança gradual de *escalas de abstração*. A Fig. 6 apresenta um exemplo parcial da variação da escala de abstração, para uma visualização baseada na notação de grafo de colaborações [Wirf 90]. O exemplo mostra o estado inicial da visualização e o estado seguinte à modificação da escala produzida pelo usuário. Neste caso a visualização é transformada automaticamente sem gerar uma nova janela.

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 da Fig. 6:

(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

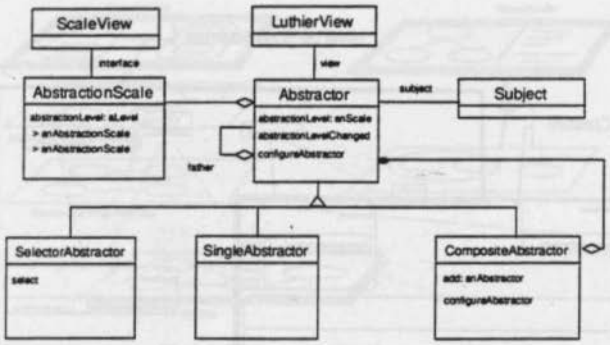

Fig. 7. Hierarquia de Classes do Sub-Framework de Abstrações

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.

3.3.1 Estrutura do sub-framework

A Fig. 7 apresenta o diagrama de objetos do sub-framework de abstrações. O mecanismo de escalas de abstração é definido pela classe *AbstractionScale,* 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 a grau de detalhe desejado na visualização corrente (a interface *default* provida é uma barra de *scrolling*).

A classe abstrata *Abstractor* define o comportamento genérico dos abstratores. Cada abstrator terá associada uma instância de *AbstractionScale,* a qual definirá o seu nível de abstração corrente; uma visão (genericamente representadas pela classe *LuthierView*) e um sujeito (representado genericamente pela classe *Subject*).

Os abstratores podem ser organizados numa hierarquia duplamente conectada de composição. A classe *CompositeAbstractor* provê o comportamento genérico de abstratores compostos. O método *configureAbstrator* é propagado aos componentes para criar representações hierárquicas de abstratores quando o sujeito do abstrator composto é fixado. Desta forma um abstrator de hierarquias de classes pode criar abstratores de classes os quais, por sua vez, podem criar abstratores de métodos.

As visões solicitam a informação a ser apresentada graficamente através de duas mensagens padrão: *getComponents* e *getRelations.* 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. O exemplo abaixo mostra a implementação de um destes métodos:

```
getComponets
        self abstractioLevel > self abstractionRepresented
                ifTrue:[ ^self getFullInformation]
```

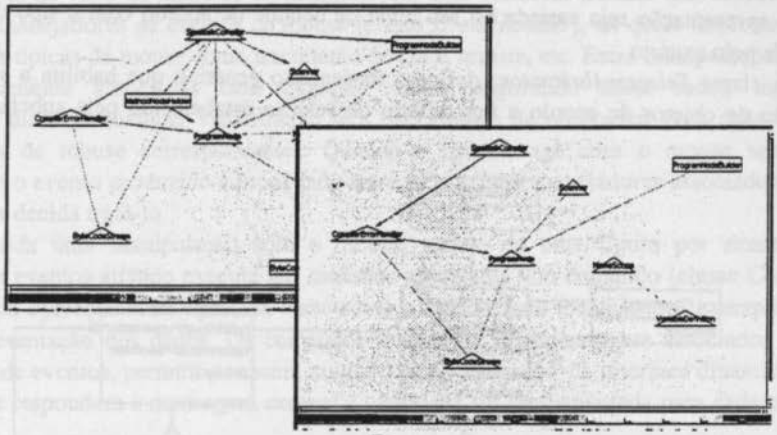

Fig. 8. Diferentes visualizações produzidas depois da instalação de seletores

```
ifFalse:[ ^self getAbstractInformation].
```

As mensagens *getFullInformation* e *getAbstractInformation* podem ser redefinidas por diferentes abstratores para retornar a informação a ser visualizada. Assim, o abstrator de subsistemas utilizado no exemplo da Fig. 6 provê a implementação seguinte:

```
getFullInformation
        "Retorna todos os seus componentes, os quais são abstratores de classes"
        ^components

getAbstractInformation
        "Retorna uma lista vazia de componentes"
        ^#().
```

Desta forma, quando o nível de abstração está no nível de subsistemas, o abstrator informa às visões que não existem componentes a serem visualizados. Quando o nível selecionada da escala é maior então o abstrator retorna a lista dos seus componentes, aparecendo em conseqüência na tela as classes que compõem o subsistema. 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 *updatePresentation*.

3.3.2 Seletores

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

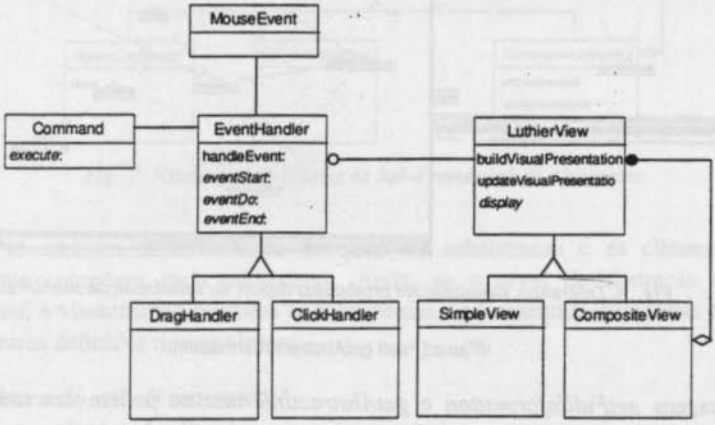

Fig. 9. Hierarquia Parcial do Sub-Framework de Interface com Usuário

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.

A Fig. 8 apresenta um exemplo de uma visualização de relacionamentos entre classes. Os triângulos representam hierarquias de classes enquanto os retângulos representam classes isoladas. As setas representam relacionamentos entre classes. No exemplo, a imagem superior mostra a visualização completa da aplicação; já na segunda a imagem o usuário solicitou só mostrar aquelas classes e relacionamentos que envolvam métodos abstratos. O seletor de propriedades agora só habilita aqueles objetos que cumprem com condição estabelecida e a visualização original se reconfigura da maneira mostrada na figura. Desta forma os abstratores habilitam um mecanismo de exploração da informação reutilizável independente das visualizações.

3.4 O Sub-Framework de Interface com o Usuário e Visualização

Este sub-framework é uma extensão do suporte gráfico provido pelo ambiente Smalltalk-80, o qual se baseia na composição hierárquica de objetos visuais [ParcPlace 95]. 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.

Do ponto de vista da interface de manipulação, uma visão em Luthier tem associados um conjunto de manejadores de eventos de mouse (classe *EventHandler*), 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 as manipulações de mouse correspondentes. Quando o usuário age com o mouse sobre uma representação o evento produzido é propagado entre os possíveis manejadores associados até que algum de eles decida tratá-lo.

Terminada uma manipulação com o mouse, arraste de uma figura por exemplo, o manejador de eventos ativado executa um *comando* associado. 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 especifica.[1]

3.4.1 Construção Flexível de Visões

Uma visão em Luthier é governada pelos abstratores, os quais requerem a construção da visão através da mensagem *build-VisualPresentation*. Cada tipo de visão deve implementar esta mensagem para construir a estrutura de objetos visuais que será desenhada na tela [2].

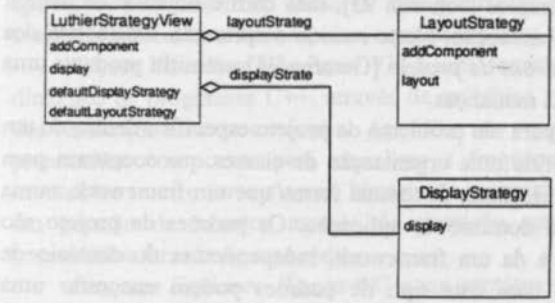

Fig. 10 Estratégias de Apresentação

Um tipo especial de visão, *LuthierStrategyView*, provê o suporte para implementar visualizações abstratas que permitem variar dinamicamente tanto a distribuição espacial como a representação gráfica da informação a ser visualizada. Uma visão deste tipo é composta por duas estratégias: uma estratégia de distribuição (*LayoutStrategy*) a qual encapsula o algoritmo que distribui os objetos na área de visualização, e uma estratégia de apresentação que define a forma em que a informação será apresentada graficamente. A visão abstrata usa estas duas estratégias propagando as mensagens que recebe para agregar componentes (*addComponent*) e para mostrar-se graficamente (*display*). Encapsulando estes dois aspectos em objetos separados é possível variá-los dinamicamente simplesmente trocando de estratégia utilizada.

[1] Esta arquitetura de interfaces é baseada no modelo proposto em [Szekely 88], quem denomina *Recognizers* aos manejadores de eventos.

[2] Em Smalltalk-80 as apresentações gráficas devem ser construídas utilizando a biblioteca de componentes visuais básicos, os quais encapsulam os mecanismos para produzir o desenho na tela.

A estratégia de apresentação genérica define um protocolo fixo para criar a estrutura de objetos gráficos que produzirão o desenho na tela. Cada tipo de apresentação deve implementar o método *buildGeometricPresentation* , o qual deve retornar uma estrutura de objetos visuais (retângulo, circulo, linha, texto, etc.) que será utilizada quando o método *display* seja chamado para produzir o desenho.

Com este suporte diferentes algoritmos de desenho e distribuição de grafos são providos por Luthier, os quais podem ser trocados dinamicamente durante a execução da ferramenta. Também o protocolo fixo estabelecido pelas estratégias permite criar novas representações gráficas sem necessidade de modificar as apresentações base utilizadas para construir uma visualização, bem como a construção de visões paramétricas cujo estrutura de desenho seja gerada por um editor de imagens.

4. Considerações Metodológicas: O papel dos padrões de projeto

As seções precedentes apresentam sinteticamente os principais aspectos de um framework para a construção de ferramentas de análise de programas orientados a objetos. Os resultados aqui apresentados são o produto de várias iterações de projeto realizadas ao longo de um ano e meio. Através deste ciclo o framework está evoluindo desde uma estrutura baseada principalmente em herança para a estrutura descrita, a qual suporta um alto grau de flexibilidade para a composição de ferramentas com pouca necessidade de programação de novos componentes.

O framework foi desenvolvido segundo o método habitual de desenvolvimento de um framework a partir da generalização de exemplos [Johnson 93], mas com o objetivo de manter um alto grau de flexibilidade em tempo de execução. Neste sentido a aplicação sistemática dos princípios de projeto introduzidos pelos *padrões de projeto* [Gamma 94] permitiu produzir uma estrutura inicial mais facilmente adaptável às mudanças.

Um *padrão de projeto* é uma solução para um problema de projeto específico dentro de um projeto orientado a objetos. Um padrão define uma organização de classes que cooperam para resolver o problema em forma extensível e elegante, da mesma forma que um framework, numa escala maior, resolve o problema para um domínio de aplicação. Os padrões de projeto são projetos reutilizáveis, numa escala menor à da um framework, independentes do domínio de aplicação. Os leitores não familiarizados com este tipo de padrões podem encontrar uma descrição detalhada no catálogo de padrões apresentado em [Gamma 94].

Por exemplo, os abstratores correspondem aproximadamente com o padrão *Proxy*, o qual introduz a idéia de objetos representantes de outros objetos, os quais podem restringir o acesso ao objeto original. Também o mecanismo de composição abstratores corresponde com o padrão *Composite* o qual descreve a forma de organizar hierarquias recursivas de objetos. O sub-framework MOM está composto por vários padrões como o *Decorator,* o qual introduz a noção de extensão de funcionalidade de um objeto através da construção de cadeias de objetos que propagam mensagens ao seu componente depois de executar alguma função adicional. Este mecanismo é utilizado para a reflexão de mensagens. Os gerenciadores de meta-objetos correspondem com o padrão *Mediator* o qual define a forma de simplificar a interação entre objetos independentes para aumentar a sua reutilizabilidade. O mecanismo de tratamento de

eventos corresponde com o padrão *Chain of Responsibility*, o qual define a forma de organizar uma cadeia de objetos encarregados de tratar alguma requisição que é propagada até que algum manejador a trate. A classe *Command* corresponde diretamente com o padrão de projeto do mesmo nome, o qual estabelece a forma de separar a interface com as operações da aplicação. Finalmente as estratégias de distribuição são uma aplicação do padrão *Strategy* o qual especifica a forma de encapsular um algoritmo que pode ter diferentes implementações de forma tal que o algoritmo a utilizar possa ser trocado dinamicamente.

5. Trabalho Relacionado

A área de ferramentas para apoio na compreensão de programas orientados a objetos esta se convertendo num foco de intensa pesquisa no últimos três anos, particularmente para apoio na visualização de programas C++.

O trabalho mais relacionado com Luthier é o framework BEE++ [Bruegge 93] que provê uma plataforma para construir ferramentas de análise dinâmica de programas C++ distribuídos. O framework suporta a construção de ferramentas distribuídas para monitoramento de eventos, visualização e depuração. O aspecto mais relevante do framework é introdução de hierarquias de eventos, as quais 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 muito importante para suportar a análise dinâmica de aplicações em diferentes contextos. A principal limitação do framework é que requer a modificação manual do código das aplicações para inserir os "sensores" que gerarão os eventos desejados pelo usuário. Desta forma, o usuário precisa conhecer *a-priori* aqueles aspectos que serão analisados.

Object Visualizer [De Pauw 93] é uma ferramenta para a visualização de comportamento dinâmico de programas C++, 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, historia de atribuição de instâncias, etc. A ferramenta se baseia em geração de eventos através de anotação automática de código, os quais podem ser transmitidos através de uma rede.

Program Explorer [Lange 95] é um protótipo de pesquisa para apoio na compreensão de programas C++, através da utilização de técnicas de visualização. O objetivo da ferramenta é prover visões centradas tanto nas classes quanto nas instâncias da estrutura e comportamento de grandes sistemas, essencialmente baseadas em grafos. A caraterística distintiva desta ferramenta é a utilização de uma representação Prolog da informação estática do programa analisado, a qual é criada e gerenciada por uma aplicação específica (Program Database). Segundo os autores, esta representação permite a especificação da estrutura do programa ou framework em diversos níveis de abstração e definir regras para filtragem de informação em tempo de execução para realizar consultas acerca da estrutura e relacionamentos tanto de classes quanto de instâncias. As visualizações providas são principalmente visualizações baseadas em grafos, para visualizar hierarquias de herança, interconexão de instâncias através de mensagens e grafos de interação de objetos [Rumbaugh 91]. Estas visões são relacionadas através de capacidades de navegação que

permitem explorar diferentes espaços de informação centrando o foco de atenção em aspectos específicos do programa.

VizBug++ é um sistema de visualização para depuração de programas C++ [Stasko 94] que integra visões diagramáticas da hierarquia de classes, instâncias e fluxo de controle. VizBug++ se baseia na interpretação de eventos para gerar uma animação da execução com setas que representam a invocação de funções e métodos. O sistema suporta funções de navegação semelhantes às de Luthier, mas é pensado para análise de programas e não de frameworks, razão pela qual não enfatiza os aspectos próprios dos frameworks como componentes abstratos por exemplo.

6. Conclusões

Nas seções precedentes apresentara-se a descrição sintética dos principais aspectos arquitetônicos do framework Luthier, o qual fornece uma infra-estrutura altamente flexível a adaptável para a construção de ferramentas de análise e visualização de programas orientados a objetos.

Uns dos aspectos mais importantes de Luthier é o mecanismo de construção de visualizações cuja semântica é determinada pela composição de abstrações. Através deste mecanismo uma visualização pode ser gradualmente transformada sob controle externo do usuário. Esta é uma caraterística não disponível habitualmente nos sistemas de visualização de software existentes.

Do ponto de vista das contribuições do projeto, as principais contribuições são o conceito de gerenciadores de meta-objetos e o suporte explícito de mecanismos de abstração para reconfiguração dinâmica de visualizações. Através da combinação adequada destes mecanismos consegue-se a implementação de estruturas altamente complexas de maneira independente das visualizações produzidas. Isto aumenta o grau de reutilizabilidade das visualizações, as quais são freqüentemente o aspecto que mais custo implica na criação de ferramentas de visualização de software.

Além da ferramenta Luthier, o framework de meta-objetos MOM, já tem demonstrado a sua versatilidade em várias aplicações diferentes, como por exemplo, suporte para interfaces baseadas em som [Balista 96] e suporte reflexivo para uma linguagem de agentes inteligentes [Amandi 96].

O mecanismo genérico de abstratores ainda se encontra em etapa de evolução. Novas aplicações poderão requerer modificar ou acrescentar funcionalidade. No estado atual, só foi testado na ferramenta Luthier, na qual ajudou a reduzir em muito a complexidade das visualizações existentes.

7. Referências

[Amandi 96] A. Amandi, A. Price, *Agent-Oriented Programming based on Objects and Logic*, submitted to Agent Theories, Architectures and Languages Workshop, Budapest, August 1996.

[Arango 94] G. Arango, *Domain Analysis Methods*, In: Software Reusability, W. Schäfer, R. Prieto-Diaz, M. Matsumoto Eds., Ellis Horwood, England, 1994. pp. 17-47.

[Balista 96] A Balista, R.T. Price, *Um Ambiente para Sonorização não Intrusiva de Aplicações Orientadas a Objetos*, UFRGS, II, CPGCC, submetido ao X SBES, também disponível em *ftp:caracol.inf.ufrgs.br/pub/amadeus.*

[Brown 84] M. Brown, R. Sedgewick. *A system for Algorithm Animation.* ACM Computer Graphics, Vol. 07. Nro. 94, pp 177-186

[Bruegge 93] Bruegge B.; Gottschalk T.; Luo B. *A Framework for Dynamic Program Analyzers*, Proceedings OOPSLA'93, Washington, D.C., October 1993.

[Campo 95a] M. Campo, R.T. Price, *Meta-Objetct Support for Framework Understanding Tools*, ECOOP'95 Workshop on Adavances on Reflection and Meta-Object Protocol, Aarus, Denmark, August 1995. Também disponível por *ftp: caracol.inf.ufrgs.br:/pub/amadeus.*

[Campo 95b] M. Campo, R.T. Price, *O Uso de Técnicas Navigacionais e Reflexivas como Apoio na Compreensão de Frameworks Orientados a Objetos,* IX Simpósio Brasileiro de Engenharia de Software, Recife, Brasil, Outubro 1995.

[Campo 96a] M. Campo, R.T. Price, *A Framework for Customizable Meta-Object Support for Smalltalk-80*, Anais do I Simposio Brasileiro de Linguagens de Programação. Também disponível por ftp: *caracol.inf.ufrgs.br:/pub/amadeus*

[Campo 96b] M. Campo, R.T. Price, *A visual Reflective Tool for Framework Understanding*, In: Technology of Object-Oriented Languages and Systems 19, Prentice-Hall, UK, February 1996.

[De Paw 93] De Paw, W.; Helm, R.; Kimelman, D.; Vlissides, J. *Visualizing the Behavior of Object-Oriented Programms*, Procs. OOPSLA'93, Washington, D.C., October 1993.

[Deutsch 89] P. Deutsch, *Design Reuse and Frameworks in the Smalltalk-80 System*, In: Software Reusability Vol. II. ACM Press, 1989.

[Gamma 94] E. Gamma, *et al*, *Design Patterns: Micro-Architectures for Reusable Object-Oriented Design*, Addison-Wesley, 1994.

[Helm 90] Helm R., *et. al., Contracts: Specifying Behavioral Compositions in Object Oriented Systems* , Procs. OOPSLA'90, Otawa, Canada, October 1990.

[Johnson 89] R. Johnson and B. Foote, *Reflective Facilities in Smalltalk-80.* OOPSLA '89 Proceedings. ACM SIGPLAN Notices Vol. 24, Nr 10, October 1989. pp. 327-335.

[Johnson 93] R. Johnson, *How to Design Frameworks*, Notes of OOPSLA'93 tutorial, Washington DC, October 1993.

[Lange 95] Lange, D.; Nakamura Y.; *Interactive Visualization of Design Patterns Can Help in Framework Understanding*, Procs. of OOPSLA'95 Conference, Austin,Texas, August 1995.

[Maes 87] P. Maes, *Concepts and Experiments in Computational Reflection*, OOPSLA '87 Proceedings ACM SIGPLAN Notices Vol. 22, Nr. 12, December 1987.

[Ortigosa 95] A. Ortigosa, *Proposta de um Modelo para Ambientes de Desenvolvimento de Software Centrados no Processo*, Dissertação de Mestrado, UFRGS,Instituto de Informática, CPGCC, Porto Alegre, Brasil, 1995.

[ParcPlace 95]ParcPlace International, VisualWorks 2.0 Reference Manual, 1995.

[Roman 93] Roman, G.; Cox, K. *Program Visualization: The Art of Mapping Programs to Pictures*. In: Proceedings Conference on Software Engeneering, 14, 1993, Melbourne, Australia., 1993. p. 412-420.

[Rumbaugh 91] J. Rumbaugh, *et al*, *Object Oriented Modeling and Design*, Prentice-Hall, 1991.

[Stasko 94] Stasko J.; Jerding, F. *Using Visualization to Foster Object-Oriented Program Understanding*. Georgia Institute of Technology, Technical Report GIT-GVU-94-33, July 1994.

[Wirf 90] Wirfs-Brooks R- Johnson R., *Surveying Current Research in Object Design*, Communications of the ACM, September 1990, Vol 33, Nº 9.