RHYCARDO LUIZ MONTEIRO

JHOTSEA: UM FRAMEWOK PARA CONSTRUÇÃO DE EDITORES GRÁFICOS SEMÂNTICOS

FLORIANÓPOLIS-SC

2003

UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM

CIÊNCIA DA COMPUTAÇÃO

RHYCARDO LUIZ MONTEIRO

JHOTSEA: UM FRAMEWORK PARA CONSTRUÇÃO DE EDITORES GRÁFICOS SEMÂNTICOS

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação.

Ricardo Pereira e Silva, Dr.

Florianópolis, fevereiro de 2003.

JHOTSEA: UM FRAMEWORK PARA CONSTRUÇÃO DE EDITORES GRÁFICOS SEMÂNTICOS

RHYCARDO LUIZ MONTEIRO

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação Área de Concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós Graduação em Ciência da Computação.

_____

Prof. Fernando A. O. Gauthier, Dr
Coordenador do Curso

Banca Examinadora

_____

Prof. Ricardo Pereira e Silva, Dr.
Orientador

_____

Prof. Roberto Tom Price, Dr.

_____

Prof. Roberto Willrich, Dr.

_____

Prof. Frank Augusto Siqueira, Dr.

"*... nele, digo, no qual também fomos feitos herança, havendo sido predestinados conforme o propósito daquele que faz todas as coisas segundo o conselho da sua vontade*."

Efésios 1:11

AGRADECIMENTOS

Agradeço a meu Senhor e Rei, por ter me concedido o dom da vida e me sustentado ao longo dela, permitindo ainda que ousasse ingressar nesse programa, apesar de todas as minhas dificuldades.

A Sheila, minha esposa e companheira, pelo carinho e paciência nesses anos. Pela renúncia aos momentos em se fez necessário estar com “o mestrado”, entendendo-o como também conquista nossa - te amo!

Aos meus filhos Felipe, Lucas e Isabela pela abnegação e compreensão com que trataram os momentos de ausência.

Aos meus pais porque acreditaram e continuam acreditando em mim, em cujo exemplo continuo a me espelhar, sinto-me honrado por poder honrá-los.

A Kedma e D.Dalva pela torcida e pela intercessão e por que fizeram dos meus sonhos também os seus. Jamais me esquecerei.

Aos pastores Osmair, Gisela, Júlio e Patrícia pois muito antes de começar nesse programa já intercediam pelo seu sucesso.

A memória de meu avô José que ao lado da avó Damiana iniciaram a saga dos Monteiros, ousou sair do cenário da escravidão e das senzalas e escolheu um novo lugar para as gerações futuras de seu clã.

A memória de minha vó Nega e do meu avô João, guerreiro obstinado que combateu a tirania alemã e, de dentro das trincheiras acreditava haver oportunidade e futuro digno para a sua descendência.

A memória do meu sogro Antônio, um idealista, que desde cedo acreditou em minha capacidade e, apesar da breve convivência me deu testemunho de firmeza na conquista dos sonhos.

Ao Prof. Ricardo, meu orientador, por ter aceitado o meu convite para orientar-me nessa dissertação. Agradeço a dedicação, o profissionalismo, a paciência e pelo célebre artigo “de boas intenções...”, que sempre me provocou nos momentos de desalento. Sinto-me honrado por herdar parte de suas expectativas e permitir que “reusasse” a sua experiência.

Aos membros da banca examinadora por terem aceitado o convite para participarem desta banca.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

CASE	*Computer aided Software Engineering* (Engenharia de Software auxiliada por computador)
DLL	*Dinamic Link Library* (biblioteca dinâmica de ligação)
GUI	*Graphics User Interface* (interface gráfica do usuário)
JDK	*Java Development Kit* (kit de desenvolvimento Java)
OOAD	*Oriented Object Analysis Design* (análise e projeto orientados a objetos)
OOPSLA	*Conference on Object-Oriented Programming Systems, Languages and Applications* (Conferência sobre Programação de Sistemas, Linguagens e Aplicações Orientados a Objeto)
UML	*Unified Modeling Language* (linguagem de modelagem unificada)

RESUMO

Os frameworks representam um marco no reuso de artefatos de software. Entretanto, a complexidade envolvida em seu desenvolvimento e uso representa um obstáculo de difícil transposição.

O uso de ambientes que contemplem as vantagens do paradigma da orientação a objetos se alia agora às ferramentas de composição gráfica a fim de conferir intuitividade e clareza à apresentação da informação.

Esta dissertação apresenta um estudo sobre frameworks, discorrendo sobre aspectos conceituais e de projeto. Este documento apresenta um framework, o JHotSea, voltado para criação de editores para gráficos estruturados que possua semântica associada e que possam ser usados em ambiente Web.

O JHotSea é implementado em Java e aproveita as vantagens da orientação a objeto com reuso intensivo a nível de código, conceitos da linguagem de implementação, projeto e nos conceitos do domínio da aplicação. Ao final, são apresentadas duas aplicações implementadas a partir desse framework, com a finalidade de elucidar melhor os conceitos e a solução apresentada.

Palavras-chave: *framework, componente, editor gráfico, linguagem Java.*

ABSTRACT

The frameworks have represented a must concerning the reuse of software products into the development based on components. However, the complexity related to its application represents a huge obstacle with a hard overcome process.

The use of environment which focus on of the paradigm of object orientations get together now with the tools of graphics composition so that intuition and clarity could be added to the information.

This document presents a study about framework conceptual and project aspects. This documents too presents a framework for creation editor of structured graphical and distributed editors on Web environment.

This document presents a study and development of a framework for creation of structured graphical and used on Web environment.

The JHotSea is implemented in Java language and utilize the advantage of object orientation with intensive reuse in code level, concepts of implementation language, project and concepts of application domain. To final, are stated two applications implemented starting of this framework, with a finality of elucidate better the concepts and present solution.

Key-words: *framework, component, drawing editor, Java language.*

1. INTRODUÇÃO

O esforço tecnológico em Engenharia de Software tem sido canalizado não só no sentido de se obter produtos de qualidade mas também está voltado para aumento de produtividade no seu desenvolvimento e manutenção. Técnicas de reuso de artefatos de software que já vinham ensaiando algum ganho em paradigmas anteriores, emergem de simples estado-da-arte para o *status* de metodologia de eficácia comprovada.

Com o surgimento da orientação a objetos abriram amplas possibilidades para otimização do reuso no processo desenvolvimento. Tornou-se possível então, criar aplicações diferenciadas estendendo estruturas de classe com implementação incompleta[Boch,97].

A elaboração de um projeto de software passa pela avaliação criteriosa de análise de domínio da aplicação e pela escolha de quais técnicas de reuso vão ser utilizadas[Riehle,00]. Isso implica na escolha e no conhecimento de variadas tecnologias não necessariamente integradas até hoje. Os frameworks tratam-se de uma técnica de reuso voltada para a construção de componentes flexíveis e genéricos para um domínio específico de aplicações, a partir do qual é possível desenvolver diferentes aplicações desse mesmo domínio, com maior produtividade[Boch,97].

O domínio do ciclo de vida do framework, associado a linguagens orientadas a objetos como o Java e as técnicas de reuso, podem conferir também maior expansibilidade aos sistemas[Gosling,95]. A manutenção desses sistemas pode ser feita de maneira mais dinâmica e segura. O advento da internet abriu novas possibilidades para uso destas tecnologias em um novo ambiente.

Juntamente com a orientação a objetos proliferou também o uso de ferramentas que manipulam objetos gráficos, pois a abstração desses elementos é de fácil visibilidade nesse paradigma. A necessidade de aplicações que façam melhor uso de recursos gráficos devido à sua clareza é cada vez maior.

O fator decisivo na proliferação de aplicações gráficas foi o aumento na capacidade do hardware uma vez que operações com gráficos exigem esforço de processamento considerável.

Aplicações de registro e apresentação de informações possuem freqüentemente interfaces gráficas embutidas. A disponibilidades desses recursos fomentou o uso de editores gráficos a fim de tratar informações expressas graficamente. Nesse cenário é que surgiu a idéia de reusar frameworks gráficos[1] permitindo reaproveitar abstrações flexíveis já existentes.

O framework Hotdraw [Brant,95] é um exemplo clássico de uso desse tipo de artefato para o desenvolvimento de aplicações que possuam ambientes de desenvolvimento ou editores gráficos embutidos . Assim, aplicações para outros fins específicos mas que pertençam ao mesmo domínio de aplicação, podem fazer uso do framework.

O comportamento desses editores no ambiente distribuído da internet ainda é alvo de estudo. A robustez das estruturas até agora existentes pode inviabilizar uma solução pois dependem de características do ambiente.

Alguns fatores, tais como a velocidade de transmissão e a eficácia dos protocolos podem degradar uma aplicação aumentando o tempo de resposta às operações. Um framework o domínio de aplicações gráficas precisa ser dotado de características que confiram leveza e flexibilidade para uso nesse editores [Johnson,92].

---

[1] framework gráfico – componente de software flexível que atua como uma espécie de núcleo a partir do qual é possível desenvolver diversas aplicações de edição gráfica. É objeto de discussão do capítulo 4.

1.1 Contexto de Trabalho

O desenvolvimento de aplicações gráficas baseadas em técnicas de reuso apresenta vantagens sobre a abordagem tradicional. Além de diminuir o tempo de desenvolvimento, reusa-se toda a qualidade do conjunto de objetos a ser reusado.

O grau de flexibilidade pode ser aumentado com um refinamento contínuo desse conjunto. Em contrapartida, a complexidade resultante desse processo pode representar um sério obstáculo para o aprendizado do seu uso[Boch,97].

O comportamento de um framework em ambientes distribuídos como a internet, que contempla uma variedade de soluções, ainda necessita de avaliação. A popularidade dos navegadores (*browser*) converge cada vez mais para o uso dos recursos da Web[2], especialmente nas aplicações que podem ser inseridas dentro de documentos HTML. As funcionalidades dessas soluções são bem conhecidas mas o comportamento no ambiente ainda conserva possibilidades não totalmente exploradas.

Assim, a especificação de frameworks e componentes[3] frente às restrições de uma aplicação do tipo *applet*[4], por exemplo, é ainda um campo aberto à exploração.

Em ambientes cada vez mais interativos e dotados de riqueza gráfica a cultura do uso de ícones avança sobre o uso convencional de textos, proporcionando interfaces gráficas cada vez mais intuitivas e o surgimento de novos editores de gráficos especializados.

Esses fatores relacionados tem conferido diversidade no desenvolvimento de softwares de edição gráfica, seja na confecção de simples gráficos, diagramas ou até em sofisticadas ferramentas de modelagem.

Considerando os aspectos descritos das dificuldades e obstáculos ao desenvolvimento orientado a frameworks e componentes, a hipótese básica em que se

---

[2] Web – Diminutivo de WWW, *Wide World Web* – padrão para páginas de hipertexto para internet.

[3] Componente – artefato de software que encapsula dados e funcionalidades (canais) que podem ser gerenciadas através de interfaces. Por serem genéricos permitem ganho de produtividade em desenvolvimento. Esse assunto é tratado no capítulo III.

[4] applet – Espécie de aplicação em linguagem Java que só pode ser executada dentro de uma *home-page*.

baseia esse trabalho é o desenvolvimento de um framework, o JHotSea, voltado para a criação de editores gráficos e, que apresente integração com o ambiente Web, usando para tal a linguagem Java. A título de comprovação foram desenvolvidas a partir dele duas aplicações.

Aplicações que possuam ferramentas de edição gráfica embutida e que necessitem ser multiplataforma poderão ser desenvolvidas a partir desse framework e o fato estar integrado ao ambiente internet lhe permite visibilidade global.

1.2 Organização do texto

O presente trabalho está organizado em sete capítulos, cujos conteúdos são descritos a seguir:

O capítulo 2 apresenta a abordagem de frameworks orientados a objetos discorrendo sobre características e formas de desenvolvimento. Este capítulo inicia apresentando o paradigma da orientação a objetos e prossegue discorrendo sobre o reuso de artefatos de software. São abordadas também questões metodológicas de análise de domínio e do desenvolvimento de frameworks.

O capítulo 3 trata a abordagem de desenvolvimento de componentes orientados a objetos. É apresentada uma discussão sobre aspectos da construção e adaptação de componentes. O capítulo finaliza referenciando as interfaces de aplicações e o contexto de sua aplicabilidade.

O capítulo 4 apresenta os frameworks gráficos HotDraw[Brant,95] e JHotDraw[Gamma,95]. A ênfase deste capítulo é apresentar suas estruturas referenciando o conjunto de funcionalidades e abstrações envolvidas, discutindo também as similaridades encontradas entre frameworks desse mesmo domínio de

aplicação. Este capítulo finaliza com a apresentação do framework MVC[5] comentando sua arquitetura e seu uso na construção de aplicações.

No capítulo 5 é apresentado o framework JHotSea, documentando os principais aspectos desta solução. Este capítulo finaliza discutindo a sua integração em ambiente internet.

No capítulo 6 discute-se a implementação de aplicações a partir do framework JHotSea. Este capítulo finaliza com a apresentação de dois exemplos de aplicação.

O capítulo 7 apresenta a conclusão deste trabalho apresentando os resultados obtidos, os fatores de limitação encontrados e a proposta de trabalhos futuros. Esse capítulo se encerra com as considerações finais.

Esta dissertação é composta ainda por dois anexos. O Anexo I procura apresentar aspectos da linguagem de programação Java relativos à implementação de aplicações de frameworks. O Anexo II apresenta alguns trechos de código escritos em Java, relativos às duas aplicações exemplificadas.

---

[5]MVC - framework *Model View Controller*, software voltado para o desenvolvimento de sistemas interativos baseados em interfaces gráficas, implementado (em Smalltalk) no ambiente VisualWorks e em outros ambientes de desenvolvimento baseado na linguagem Smalltalk. Mais que um framework específico, MVC caracteriza um estilo arquitetônico de organização de aplicações OO. É descrito na seção 4.5.

2. FRAMEWORKS ORIENTADOS A OBJETOS

A tecnologia da orientação a objetos no desenvolvimento de softwares vem sendo aplicada deste a década de 70, quando já se estudava este paradigma, especialmente em técnicas de escrita de código. Entretanto, o uso de abstrações em nível mais elevado só influenciou o desenvolvimento de software muito tempo depois. A utilização de técnicas de reuso tem sido objeto de ampla discussão e já apresenta ampla disseminação. Modelos de componentes, frameworks e padrões[6] apresentam um ganho substancial em qualidade e tempo de desenvolvimento. [Silva,00]

Abordaremos neste capítulo aspectos relacionados à construção de frameworks numa perspectiva evolutiva, procurando eliciar[7] os aspectos mais relevantes, começando do paradigma da orientação a objeto.

2.1 O paradigma da Orientação à Objeto

O termo "orientado a objetos" significa a organização do software como uma coleção de objetos discretos que incorporam tanto a estrutura como o comportamento dos dados. Esta abordagem difere da abordagem estruturada, onde a estrutura e o comportamento dos dados têm pouca vinculação entre si.

---

[6] Padrões – tradução em português de *patterns* para se referir a fragmentos de software que representam soluções comuns e aplicáveis em situações similares[Boch,97]. É abordado na seção 2.2 deste capítulo.

[7] Eliciar – esse verbo não existe na língua portuguesa, mas foi criado e tem sido utilizada por vários autores (e.g., Leite em [Leite,94]) englobando o significado dos verbos eliciar (fazer sair, extrair, trazer à tona a verdade), clarear, extrair e descobrir.

Origem da Orientação a objeto

O uso da orientação a objeto como metodologia básica para o desenvolvimento de sistemas abrangendo todo o ciclo, desde a análise até a construção de códigos, é uma prática bem recente. Apenas na década de 80 é que surgiram os primeiros estudos sobre o uso da Orientação a Objeto para especificação em projetos de sistemas [Booch,96].

A utilização destes mesmos conceitos e mecanismos para a análise de sistemas se tornou frequente e deu origem a diversos ensaios a respeito foram publicados. Sucedeu-se também a publicação de trabalhos sobre essa técnicas de modelagem para especificação de sistemas [Booch,96][Coad,92].

Características do Paradigma

O grande diferencial deste paradigma é a facilidade como que este se relaciona com o universo real. Assim, cada objeto conceitual corresponde a um objeto real em suas características e comportamento. São características comportamentais desejáveis de um objeto [Coad,92]:

Identidade - Significa que os dados são quantificados em entidades discretas e distinguíveis denominadas objetos. Estes objetos podem ser concretos ou conceituais. Cada objeto tem identidade própria, ou seja, dois objetos são diferentes, mesmo que seus dados e suas operações sejam idênticos.

Classificação - Significa que objetos com dados e operações semelhantes podem ser agrupados em classes. Uma classe é uma abstração que descreve características importantes para uma aplicação e ignora o resto. Um objeto é uma instância de sua classe, e identifica a que classe pertence.

Polimorfismo - Significa que uma operação pode ser implementada de maneira diferente em classes diferentes. Uma operação é uma ação ou transformação que um objeto realiza, ou a qual é submetido.

Um método é uma implementação específica de uma operação. Cada objeto possui informações de como executar suas operações.

Herança - É o compartilhamento de atributos e operações entre classes, baseado em um relacionamento hierárquico. Uma classe pode ser definida de maneira abrangente, e depois ser refinada em subclasses.

Os elementos de uma superclasse não precisam ser repetidos em suas subclasses, que automaticamente herdam estes elementos. A reutilização que isto proporciona é uma das principais características da orientação a objetos [Coad,92].

2.2 Reusabilidade no Desenvolvimento de Software

O reuso de artefatos de software constitui um dos fatores mais atrativos da orientação a objetos por conferir ganhos ao processo de desenvolvimento. As vantagens vão além da herança e do polimorfismo, mas herda-se também a qualidade dos projetos e códigos construídos dentro de critérios de níveis elevados da engenharia.

As premissas deste paradigma se constituem num marco conceitual para a produção de código reutilizável por meio de classificação taxonômica para as classes. Os ambientes de desenvolvimento que dão suporte a esse paradigma apresentam índice elevado de interatividade.

A figura 2.1 apresenta a evolução do reuso relacionando-a com o nível de abstração e apresenta os mecanismos de software que o implementam [Bellur,98].

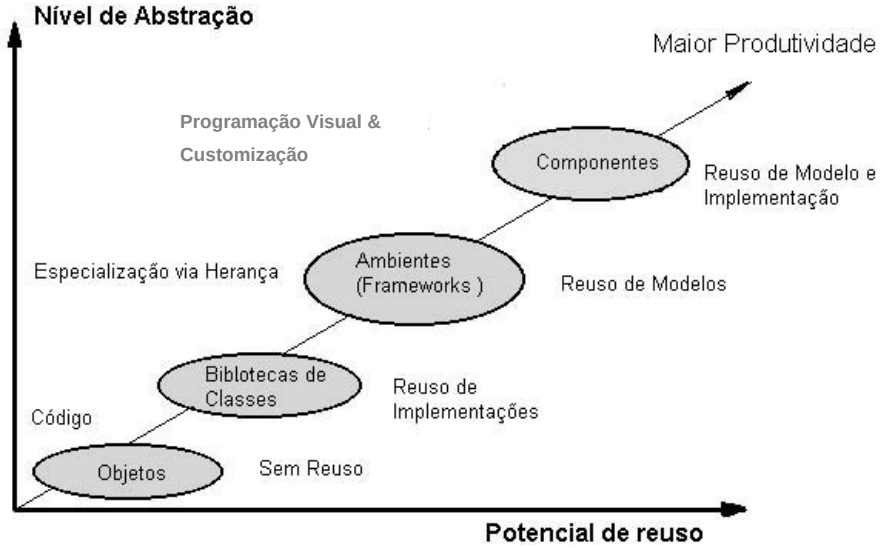


Figura 2.1: Evolução da abstração e do reuso [Bellur,98].

Reuso em projetos

A reutilização de projeto pode ser entendida como a reutilização de abstrações em alto nível (acima de código), e representa a divisão de um domínio de aplicação em componentes funcionais e os protocolos de colaboração entre eles.

A interação entre componentes é definida pelos protocolos, nas chamadas a serviços ou métodos, para executar as diferentes funcionalidades do sistema. O conjunto de serviços que um componente define, representa a sua interface externa, a qual é conhecida pelos outros componentes.

Este conjunto de classes pode representar um projeto completo de uma aplicação ou subsistema, ou simplesmente construções de alto nível que resolvem situações comuns de projeto. No primeiro caso constituem frameworks, no segundo *patterns.*

Design Patterns

Os *Design Patterns* originaram-se a partir do trabalho do arquiteto Christopher Alexander, do final da década de 70, que apresentou em seus dois trabalhos "*A Pattern Language*" [Alexander,77] e "*A Timeless Way of Building*" [Alexander,79] conceitos, exemplos e ainda propôs uma forma de documentação dos mesmos. Essa arquitetura ganhou destaque com o *workshop* de Beck durante a OOPSLA/87 sobre o seu uso na construção de janelas em *Smalltalk*[8] [Beck,87].

O padrão (*pattern)* é uma micro-arquitetura de elementos que ocorrem repetidamente, como mostra a figura 2.2. Assim, ele pode representar uma solução para um problema que ocorre com freqüência durante o desenvolvimento de software, podendo ser considerado como um par "problema/solução" [Buschmann, 96].

[8] Smalltalk – linguagem de programação orientada à objeto, criada pela XEROX-PARC em 1963.

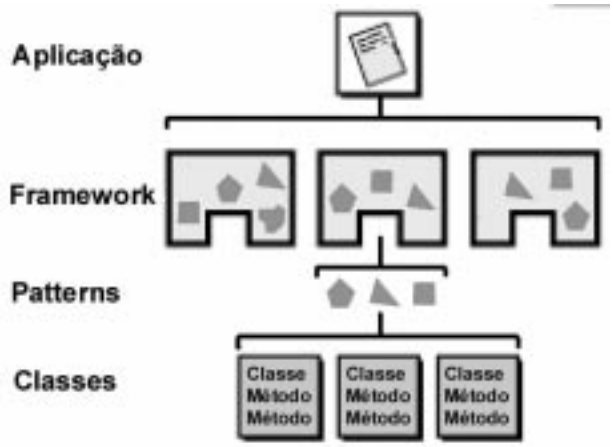


Figura 2.2: Reuso de artefatos [Taligent,94].

Projetistas familiarizados com determinados padrões podem aplicá-los de imediato em problemas de projeto, sem ter que redescobri-los [Gamma,95].

Diversos autores têm contribuído com a apresentação de padrões, variando de acordo com a escala e abstração e que deram origem a catálogos que auxiliam no seu reconhecimento.

O catálogo representado na tabela 2.1, foi proposto por Gamma, e considera dois critérios para classificação: o escopo – que é a forma de relacionamento com a classe, e o propósito – que define a abstração propriamente dita [Gamma,95].

		Propósito		
		De Criação	**Estrutural**	**Comportamental**
Escopo	**Classe**	Factory Method	Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweygth Proxy	Chain of Responsability Command Iterator Mediator Memento Observer State Strategy Visitor

Tabela 2.1: Catálogo de *Patterns* [Gamma,95].

Os padrões *Composite* e *Observer*, serão discutidos no capítulo 4 e 5 deste trabalho e as particularidades de sua implementação em Java constam no Anexo I.

Os Frameworks

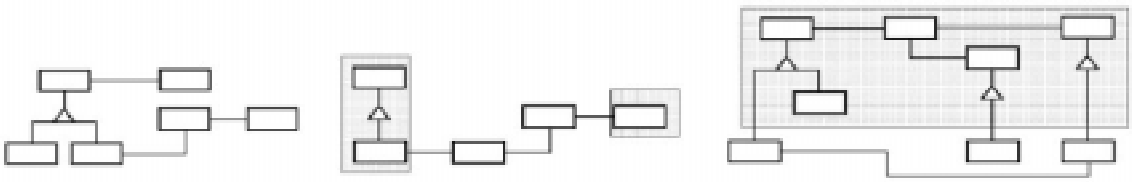
O conceito e a aplicação de frameworks evoluíram com o uso de bibliotecas de classes nos anos 80. Segundo Coad[Coad,92], trata-se de um esqueleto de classes, objetos e relacionamentos agrupados para construir aplicações específicas.

Um framework é um desenho reusável do programa ou parte do programa representado por um grupo de classes [Johnson,92].

Os frameworks são compostos por artefatos interligados (classes) e diferem das bibliotecas de classe pois os artefatos destas necessitam ser interligados pelo desenvolvedor [Silva,00] e possibilitam reutilizar não só componentes isolados, como também toda a arquitetura de um domínio específico.

A finalidade básica de um framework é ser reutilizada na produção de aplicações distinta. Pode ser visto, em termos de abstração, como uma descrição genérica aproximada das estruturas de um domínio de aplicações.

A figura 2.3 apresenta uma visão comparativa dos processos de desenvolvimento de aplicações[Silva,00].



a) usando classes b)reutilizando biblioteca de classes c)reutilizando framework

Figura 2.3: Processos de desenvolvimento de aplicações [Silva,00].

Um framework permite o reuso de projeto porque contém algoritmos abstratos e descreve a interface que o programador deve implementar e as restrições a serem satisfeitas pela implementação. Ele reusa código porque torna mais fácil desenvolver uma biblioteca de componentes compatíveis e porque a implementação de novos componentes pode herdar grande parte de seu código das superclasses abstratas [Masiero,01].

Um exemplo clássico de framework é o HotDraw, que tem sido reusado com bastante freqüência na criação de editores gráficos apenas estendendo algumas de suas

classes e acrescentando outras mais, como é o caso do SEA [Silva,00] e do EDI [Pompermaier,97].

A maioria do frameworks existentes foi desenvolvida para domínios técnicos como por exemplo os frameworks MacApp, específico para aplicações Macintosh, o Lisa Toolkit, o Smalltalk MVC, o Interviews, etc. Já os frameworks para aplicações específicas costumam não ser de domínio público, como por exemplo o ET++, JFC e da Sun, o MFC e DCOM da MicroSoft [Masiero,01].

Frameworks x Padrões

Johnson [Johnson,97] afirma que padrões são mais abstratos do que frameworks. Os padrões são menores do que frameworks, em geral são compostos por 2 ou 3 classes, enquanto os frameworks envolvem um número bem maior de classes, podendo englobar diversos padrões.

Os padrões são menos especializados do que frameworks, já que os frameworks geralmente são desenvolvidos para um domínio de aplicação específico enquanto os padrões não possuem domínio de aplicação definido. Diferentemente dos frameworks, mesmo os padrões mais especializados não são capazes de determinar a arquitetura de uma aplicação [Masiero,01][Silva,00]. No entanto, é possível reusar um padrão em um domínio de aplicação diferente e em ambientes completamente diferentes, enquanto que um framework não pode ser utilizado em outro contexto (domínios, linguagens, sistemas operacionais, etc).

2.3 Análise de Domínio

O framework é a generalização de um modelo de domínio especifico de aplicação e por essa razão deve conter a definição das classes que serão instanciadas para a construção da aplicação. O desenvolvedor tem autonomia limitada nesse processo pois o framework projetado pode conter classes que necessitem ser obrigatoriamente instanciadas ou ainda métodos que não possam ser sobrepostos. Este aspecto revela a importância da Análise de Domínio, pois deve capturar descrições que generalizem um conjunto de aplicações com similaridades de características.

A Análise de Domínio é uma etapa fundamental para a criação de artefatos de software reutilizáveis e deve capturar a essência das funcionalidades de um domínio[Prieto-Diaz,87].

Para Neighbors, a Análise de Domínio é o processo de busca por tentativas, para identificar objetos, operações e relações que especialistas de um domínio percebem ser importantes para a descrição deste domínio[Neighbor,89].

A Análise de Domínio pode ser desenvolvida em três etapas baseadas na Engenharia de conhecimento: a identificação das fontes de conhecimento, a aquisição do conhecimento existente e a representação desse conhecimento num modelo de domínio [Arango,91].

A figura 2.4 ilustra as etapas que compõem o processo de análise de domínio.

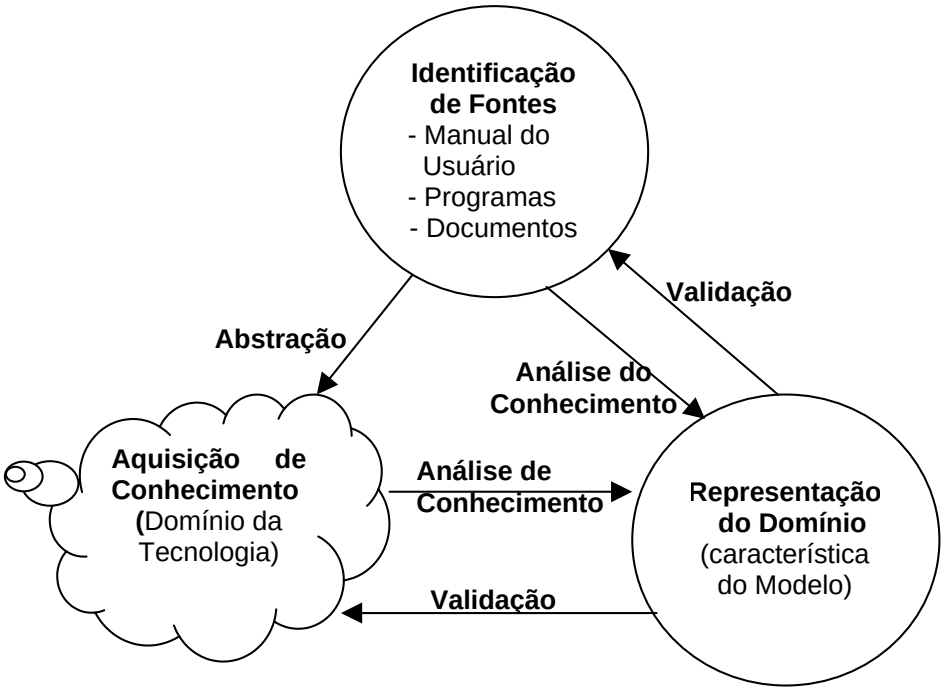


Figura 2.4 Fases do processo de Análise de Domínio.

Diversas metodologias tem sido propostas para abordar a Análise de Domínio de forma prática.

Metodologia de Henniger- o modelo proposto por Henniger consta de um repositório especificações (projeto, análise e códigos) e um hipertexto com informações de aplicações do mesmo domínio já desenvolvidas, e que pode ser atualizado para

acumular conhecimento. O hipertexto permite a organização e a localização de informações necessárias [Henninger,95].

Metodologia baseada em Ontologias formais – É uma metodologia baseada na representação de conceituação – ontologia, segundo Guarino [Guarino,1997]. Assim, a análise do domínio passa pela construção de ontologias. Consiste em um ciclo interativo de fases de planejamento, seguido da aquisição de dados (captura de ontologias), pela construção de um modelo de domínio, fazendo uso de uma linguagem formal e encerra-se com a integração com as ontologias existentes em busca de novos conceitos (reuso de conceitos)[Poli,01]. Apesar de necessidade de compreensão clara de conceitos filosóficos envolvidos, essa metodologia só pode ser aplicada utilizando-se repositório de conceitos que podem ser gerenciados por sistemas especialistas. É possível gerar especificação de componentes formais específicos a partir das ontologias do domínio[Guizzardi,01].

Metodologia de Gomaa – Nesta metodologia, a produção de modelos de domínio é tratado num ambiente com técnicas independentes do modelo tratado. Os requisitos são selecionados por um sistema especialista e em seguida a geração de especificações é automática. Esta metodologia possui formato semelhante ao das metodologias de análise como OOAD e Fusion. Com a utilização de um ciclo de vida evolutivo, a geração de aplicações pode produzir elementos que proporcionem a evolução do modelo de domínio. O modelo de Gomaa é fortemente ligado ao paradigma da Orientação à Objeto[Gomaa,94].

Metodologia de Neighbor - Também conhecida como paradigma de Draco, também é voltada para ambiente de geração de especificação. Segundo esse paradigma, produzir uma aplicação consiste em escrever uma especificação utilizando a linguagem do domínio da aplicação, que posteriormente pode ser traduzida em uma linguagem de programação convencional . Seu modelo de domínio consta dos tipos: domínio executável que corresponde às linguagens de programação, o domínio de modelagem que é o encapsulamento do conhecimento necessário para gerar uma determinada funcionalidade na aplicação e o domínio de aplicação propriamente dito que é o conhecimento capaz de produzir aplicações completas [Silva,00].

A metodologia de Neighbor possui um agravante para utilização de seu ambiente por depender da experiência humana, o que pode demandar um certo esforço em treinamento.

Das metodologias apresentadas, esta ultima é a que apresenta o mais alto nível de abrangência com relação às etapas do ciclo de vida. Produz um modelo de domínio que contém especificações e código e permite a reutilização também ao nível de especificações.

2.4 Uso dos Frameworks

Um framework, por concepção, deve possuir pontos notáveis a fim de orientar a sua implementação. São eles: [Buschman,96] [Silva,00]:

Hot spot - São ponto de especialização que representam as partes do framework de aplicação que são específicas de sistemas individuais, projetados para serem genéricos permitindo a adaptação às necessidades da aplicação.

Frozen spot - São pontos fixos que definem a arquitetura geral de um sistema de software. Seus componentes básicos e os relacionamentos entre eles permanecem fixos em todas as instanciações do framework de aplicação.

A identificação e a definição dos pontos de estabilidade(*frozen spots*) e variabilidade (*hot spots*) em *frameworks* conferem flexibilidade a ele, evitando a necessidade de reprojeto e reimplementação em cada nova aplicação desenvolvida sob esse framework.

Esses pontos são implementados pelos métodos: *Hook* – implementa *hot spots* e *template* – que implementa os *frozen spots* [Pree,94][Silva,00].

A identificação de métodos *template* e *hook* é, então, a separação e o encapsulamento de aspectos fixos e variáveis de um *framewor*k, que variam e evoluem independentemente.

Inversão de controle

Uma característica fundamental dos frameworks é a inversão de controle, ou seja, os métodos definidos pelo usuário para especializá-lo são chamados de dentro do próprio framework, e não chamados a partir do código de aplicação do usuário [Johnson 88].

O framework coordena e seqüência as atividades da aplicação. Essa característica capacita o framework para servir como esqueleto extensível. Os métodos fornecidos pelo usuário especializam os algoritmos genéricos definidos no framework para uma aplicação específica [Johnson,88] [Masiero,01].

Tipos de frameworks

Os frameworks podem ser classificados em três tipos [Johnson,88][Silva,00]:

Caixa Branca (*white-box*) **–** o reuso é provido por herança, onde usuário cria subclasses das classes abstratas do framework para criar aplicações específicas. Entretanto é necessário conhecer aspectos funcionais do framework para poder usá-lo. É mais simples de ser projetado porque não necessita ser totalmente definido, entretanto precisa de maior esforço de implementação.

O esquema da figura 2.5 apresenta a implementação R3 do *hot spot* R. É importante notar que a implementação está externa ao framework.

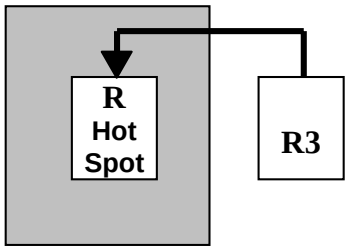


Figura 2.5: Framework tipo White-Box [Schmid,97].

Caixa-Preta (*black-box*) - o reuso é provido por composição, onde o usuário necessita combinar diversas classes concretas do framework para obter a aplicação. Assim, ele deve entender apenas a interface para poder usá-lo. È de uso mais fácil. Na maioria dos casos, é preciso selecionar as implementações necessárias.

A figura 2.6 mostra um framework tipo *caixa-preta* em que um *hotspot* R, é implementado em três alternativas: R1, R2 e R3. É relevante observar que as implementações fazem parte do framework.

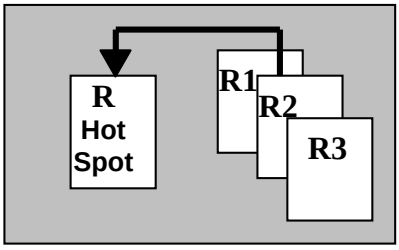


Figura 2.6: Framework tipo Black Box [Schmid,97].

Um framework caixa branca podem evoluir para se tornar um caixa preta, com uma diminuição do número de objetos a serem criados, com conseqüente aumento a complexidade das suas interconexões [Johnson,97].

Caixa-Cinza(*gray-box*) – São frameworks que apresentam comportamento híbrido, com características tanto do caixa-preta quanto do caixa-branca [Silva,00]. Caixa-preta e caixa-branca são extremos conceituais, na prática o caixa-cinza é o que ocorre com maior frequência[9].

Em frameworks caixa-branca, o uso de herança permite a visibilidade de aspectos internos das superclasses nas suas subclasses, pois "herança quebra encapsulamento" [Johnson,88]. Assim, frameworks caixa-branca são mais difíceis de usar que caixa-preta, uma vez que o desenvolvedor necessita de maior conhecimento sobre a implementação do framework. O uso de herança para combinar funcionalidade pode levar a uma proliferação de subclasses, aumentando a complexidade de software.

Frameworks caixa-preta são mais difíceis de projetar, uma vez que a interface para conexão entre componentes deve ser elaborada com cuidado, prevendo usos futuros, já que interfaces não bem elaboradas podem limitar as oportunidades de reutilização dos frameworks.

---

[9] Levantamento realizado por Yassin e Fayad em 1999 constatou: 55% dos frameworks caixa-cinza 30% dos frameworks caixa-branca e 15% dos frameworks caixa-preta[Fayed,99].

Desenvolvimento de Frameworks

O processo de desenvolver um framework consiste essencialmente em definir uma estrutura de classes que tenha a capacidade de adaptar-se a um conjunto de aplicações diferentes pertencentes ao mesmo domínio.

Um bom projeto de framework requer uma criteriosa identificação das partes que devem ser mantidas flexíveis.

Dotar um framework de alterabilidade e extensibilidade resulta na flexibilidade deste projeto, e o torna eficiente para aplicações de mesmo domínio, entretanto a característica mais importante a ser buscada é mesmo a generalidade [Silva,00].

Ciclo de Vida do Framework

Os artefatos envolvidos nos ciclo de vida do framework estão agregados uns aos outros, o que difere de uma aplicação convencional que pode representar artefatos de software isolados.

A figura 2.7 apresenta sob a forma de diagrama a ligação do framework com as fontes geradoras de informação necessárias à sua concepção.

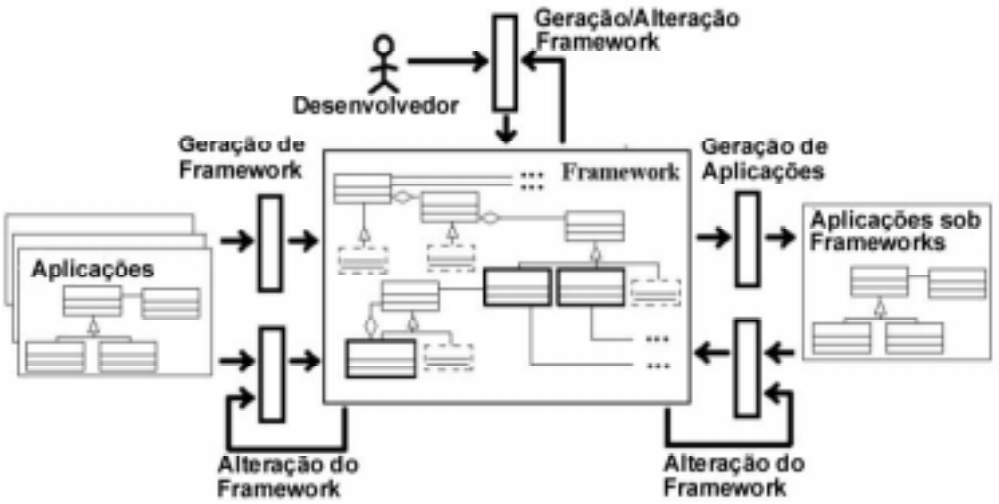


Figura 2.7: Ciclo de Vida do framework [Silva,00].

Observe na parte superior que o desenvolvedor atua na seleção das classes bem como também na sua manutenção. Definir responsabilidades e grau de flexibilidade esperada também são tarefas do desenvolvedor.

À direita do framework está o domínio composto de aplicações cuja generalização de estruturas deve ser capturada. A geração automática de frameworks é uma possibilidade, embora não se tenha conhecimento, na atualidade, de ferramentas que o façam.

Metodologias de Desenvolvimento de Frameworks

As metodologias para desenvolvimento de framework buscam definir estratégias eficazes para a construção de projetos com excelência. Apresenta-se a seguir três metodologias consideradas de referência [Silva,00] :

Projeto dirigido por exemplo - É construído a partir de abstrações obtidas a partir do exame de exemplos concretos e requer o aprendizado a respeito deste domínio. A generalização é realizada capturando a semelhança funcional de elementos agrupando-o em classes similares.

É desenvolvido em três fases: a) a análise de domínio onde se buscam exemplos e assimilam-se abstrações; b) o projeto da hierarquia de classes, otimizado com o uso de padrões e c) teste do framework reproduzindo os exemplos originais com o seu uso [Johnson,93].

O inconveniente desse método é a insuficiência de técnicas e ferramentas para sua implementação, o que sugere ao desenvolvedor a lançar mão de técnicas de OOAD para complementar o processo.

Projeto dirigido por *Hot Spot* – É baseado na criação de partes do framework flexíveis - os *hot spots*. A identificação dos *hot spots* representa a essência dessa metodologia.

A seqüência proposta por Pree prevê 4 etapas: a) identificação das classes para definição da estrutura; b) identificação dos *hot spots*, agregando em classes abstratas dados(atributos) ou funcionalidades(métodos) desde que sejam comuns às classes; c) reelaboração do projeto a fim de conferir flexibilidade ao modelo e d) refinamento e avaliação do framework [Pree,94].

Outra característica importante desta metodologia é a importância da intervenção de especialista do domínio praticamente em todas essas fases, o que confere adequabilidade ao modelo [Pree,94].

Projeto pela metodologia Taligent – Foi desenvolvido pela extinta Taligent e difere dos anteriores basicamente em dois pontos: busca desenvolver um conjunto de frameworks menores e mais simples para desenvolver aplicações o que os tornam mais flexíveis e reutilizáveis e disponibilidade de classes concretas para uso direto e a minimização de elementos (classes e métodos) a serem criados ou sobrepostos.

O desenvolvimento nesta metodologia também foi organizado em quatro etapas: a) identificação e caracterização do domínio do problema; b) definição da arquitetura e do projeto; c) implementação do Framework e o c) desdobramento do framework [Taligent,94].

O próximo capítulo prossegue no assunto reuso na abordagem voltada para o uso de componentes.

3. DESENVOLVIMENTO BASEADO EM COMPONENTES

A necessidade de construção de softwares utilizando famílias de componentes reusáveis já era conhecida durante a "crise do software", e foi recomendada em 1968 durante a Conferência de Engenharia de Software da OTAN. A abordagem para o desenvolvimento de software como um conjunto de componentes reusáveis tipo caixa-preta foi proposta por McIlroy [McIlroy,68]. Em 1976, DeRemer propôs o desenvolvimento em módulos separados e interligados posteriormente [DeRemer,76].

Posteriormente, SZYPERSKI ampliou o conceito de componentes:

"*o que torna alguma coisa um componente não é uma aplicação específica e nem uma tecnologia de aplicação específica. Assim, qualquer dispositivo de software pode ser considerado um componente, desde que possua uma interface definida. Essa interface deve ser uma coleção de pontos de acesso a serviços, cada um com uma semântica definida.*" [Szyperski,97]

3.1 Características dos Componentes

Um modelo de componentes aceitável pressupõe unidades com interfaces devidamente especificadas e com independência de tecnologia de implementação. Os pontos de acesso contidos na interface são conhecidos como canais de comunicação.

A interface então é a parte externamente visível de um componente que permite comunicação bidirecional. Ela pode apresentar um ou mais canais para conexão com outros componentes [Silva,99].

A total compatibilidade funcional e de interfaces dos componentes são algumas das dificuldades mais comuns a serem superadas no reuso de componentes. Além disso, mecanismos de identificação que permitam a seleção de funcionalidades a serem utilizadas são fundamentais no projeto de componentes.

3.2 Descrição de componentes

A descrição de um componente passa necessariamente pela descrição de suas interfaces, o que resulta apenas numa visão externa. Para esse caso, a descrição precisa conter a assinatura dos métodos oferecidos. Entretanto, para o caso de comunicação bidirecional, é necessária ainda uma descrição dos métodos requeridos.

Na proposta de Murer, a descrição dos métodos requeridos deve ser realizada em três níveis: o da interface- pelo uso de uma linguagem de especificação, o do originador – apresentando as restrições do criador do componente, e o semântico- que é a descrição mais completa das funcionalidades [Murer,96].

Especificação estrutural da interface

Essa especificação relaciona os métodos oferecidos e requeridos e a sua relação com cada canal da interface. A figura 3.1 apresenta um exemplo de três componentes, dois métodos requeridos e dois fornecidos. Note a existência de dois canais de comunicação.

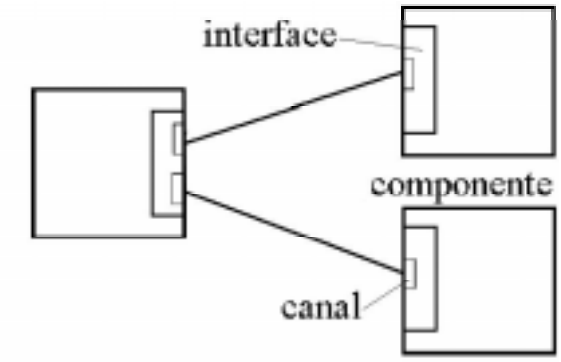


Figura 3.1 Arquitetura de componentes[Silva,00].

Silva propõe que a necessidade de uma descrição complementar à especificação para indicar a disponibilidade de cada um dos métodos nos canais. Isso pode ser feito como mostra a Tabela 3.1 [Silva,00].

Métodos / Canais	Requeridos		Fornecidos	
	mX	**MY**	**m1**	**m2**
Ca	✓	✓	✓	
Cb	✓			✓

Tabela 3.1: Especificação estrutural de componente[Silva,00].

Especificação Comportamental da interface

As redes Petri são diagramas usados para representar as restrições de ordem nas transições e podem ser usadas para descrever o comportamento das interfaces[Petri,62]. Sua simplicidade pode permitir a análise da arquitetura do componente e a verificação da ocorrência de *deadlock´*s, que são aquelas transições em que determinados métodos nunca são chamados, conforme propôs Silva. A figura 3.2 apresenta um exemplo de ocorrência de *deadlock* através de rede Petri, note [Silva,00].

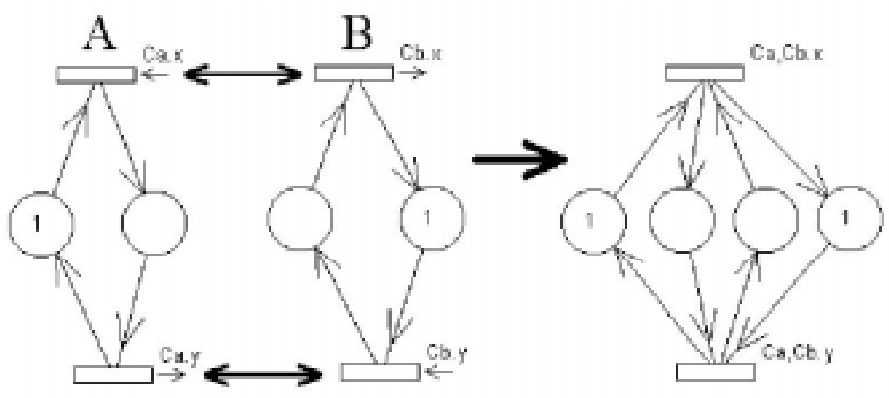


Figura 3.2: Redes Petri: ocorrência de *deadlock* [Silva,00].

3.3 Adaptação de Componentes

Um componente normalmente precisa ser adaptado aos requisitos do sistema que dele fará reuso. A questão é que suas interfaces na maioria das vezes restringem o acoplamento por serem incompatíveis com o sistema modelado, necessitando portanto serem submetidas à técnicas que removam essas restrições. Essa adaptação pode ser realizada nas seguintes abordagens [Silva,99]:

Empacotamento (*wrapping*) – consiste em criar uma visão externa diferente para o componente, ao invés de modificá-lo[Bosch,97]. Como observamos na figura 3.3, a interface de C adapta as características da interface B para que possa ser acoplada a A.

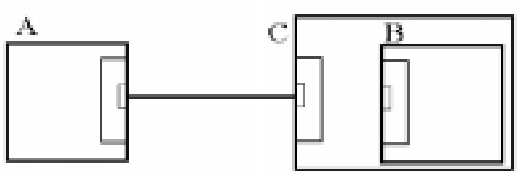


Figura 3.3: Empacotamento (*wrapping*)[Silva].

Colagem (*glueing*)- consiste em criar um componente intermediário para intermediar a comunicação. Esse componente é a cola, conforme a figura 3.4. O fator restritivo ao uso de colas é a complexidade da estrutura de programação que está oculta sob o componente [Szyperski,97].

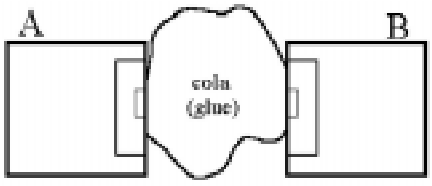


Figura 3.4: Colagem (*glueing*) [Silva,00].

Reflexão – consiste na criação de um elemento externo para monitorar a o comportamento do componente, conforme pode ser visualizado pela figura 3.5. Se por alguma circunstância um dos componentes seja impedido de passar mensagens o outro o

fará. Para que isso seja possível, é importante que A e B sejam estrutural e comportamentalmente compatíveis [Assmann,97].

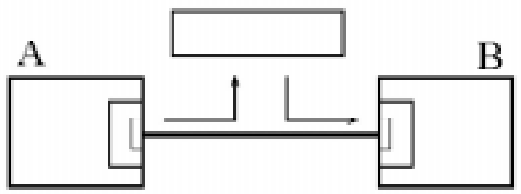


Figura 3.5: Reflexão[Silva,00].

Arcabouço de Componente (*component framework*) – trata-se de um componente que prevê o acoplamento de outros componentes. Logo, para altera-lo basta substituir os componentes que estão acoplados a ele[Helton,98] [Weck,96]. Na figura 3.6 nota-se a existências de canais que definem o comportamento do componente disponível para acoplar diferentes componentes.

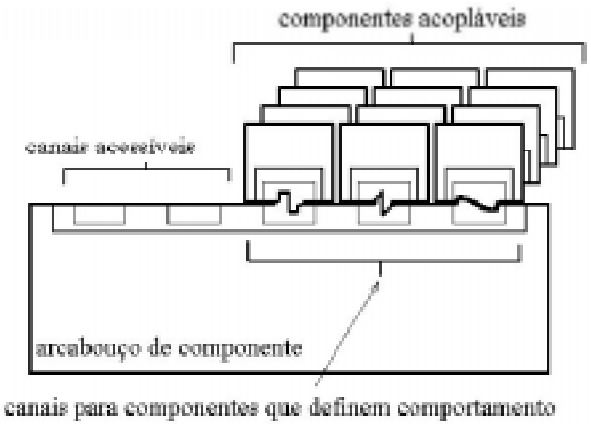


Figura 3.6: Arcabouço de componentes. [Silva,00b]

3.4 Organização de Componentes

O entendimento da abstração do componente é essencial para o seu reuso eficaz no desenvolvimento de aplicações. É igualmente importante o conhecimento e a aplicação de recursos de modelagem, pois estes contribuem para o conhecimento, projeto e uso dos componentes.

Segundo Booch, os componentes podem ser organizados, do ponto de vista da aplicação, em três tipos diferenciados [Booch,99]:

Componentes de implantação – que são aqueles necessários e suficientes para formar um sistema executável, tais como bibliotecas dinâmicas (DLL) e executáveis. Esse conceito comporta ainda modelos clássicos de objeto como COM+[10], CORBA[11] e *Java Beans*[12] e alternativos como as páginas Web dinâmicas .

Componentes de produto de trabalho – são subprodutos do processo de desenvolvimento tais como códigos-fonte e arquivos de dados gerais.

Componentes de execução – esses componentes são criados como conseqüência de um sistema em execução, como por exemplo um objeto CORBA instanciado a partir de uma DLL.

3.5 Desenvolvimento com API

Uma API – *Application Program Interface,* é essencialmente um conjunto de interfaces publicáveis e que podem ser referenciadas pelo desenvolvedor sem a necessidade de conhecer sua localização do componente que a originou, desde que ele exista.

Modelagem de API´s

As operações associadas a qualquer API projetada adequadamente serão bastante extensivas, não sendo necessária a visualização de simultânea de todas as operações. A tendência é manter as operações na base dos modelos e usar apenas as interfaces para manipular esses conjuntos de operações.

---

[10] COM+ - *Component Object Model* - Arquitetura para comunicação em sistemas distribuídos proprietária da Microsoft.

[11] CORBA – *Common Object Request Brocker* - Arquitetura que permite invocar métodos e objetos remotamente com suporte multilinguagem.

[12] Java Beans – Padrão para componentes Java para diversos tipos de aplicações.

Segundo Booch, para modelar uma API é necessário[Booch,99]:

- identificar as costuras programáticas existentes no sistema e modelar cada costura com uma interface, colecionando atributo e operações que a constituem;
- publicar apenas as propriedades relevantes da interface para identificar sua assinatura;
- modelar as funcionalidades de cada API somente quando necessário apresentar sua configuração específica.

Uso de API´s

Para construir modelos executáveis usados essas API´s é importante que as ferramentas de compilação possam identificar suas propriedades.

A figura 3.7 exemplifica a modelagem de um componente, em notação UML, que faz uso de quatro API´s publicadas.

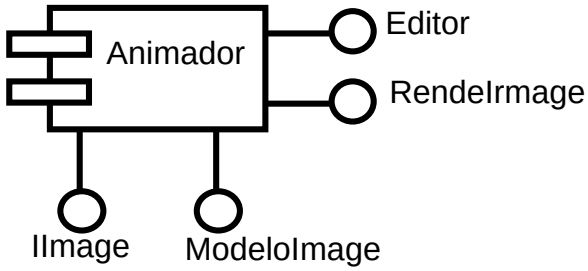


Figura 3.7: Modelagem de uma API

A maioria dos sistemas operacionais da atualidade não apenas dá suporte a publicação de API´s como mantém uma extensa lista de API´s publicadas que fazem uso de componentes do próprio sistema operacional.

Evidentemente que o aspecto predominante nas definições relativas ao seu uso é o conhecimento do domínio da aplicação.

O próximo capítulo descreve alguns dos frameworks gráficos mais usuais e procuram elicitar seus respectivos domínios de aplicação.

4. FRAMEWORKS GRÁFICOS

Os frameworks gráficos representam um domínio de aplicação usual em qualquer aplicação que inclua em suas funcionalidades a manipulação de figuras. A partir da década de 80, com a popularização dos ambientes gráficos essa característica tem caráter de obrigatoriedade em softwares que desejam comunicar informações graficamente.

Este capítulo procura apresentar os dois frameworks clássicos utilizados na criação de editores gráficos: o HotDraw [Johnson,92] e o JHotDraw [Gamma,95], escolhidos pelas suas características e similaridades. Outros casos pertencentes ao mesmo domínio serão citados mais sucintamente a título de comparação. Esse comparativo é importante, pois auxilia na definição do projeto, além de contribuir para o entendimento do domínio da aplicação.

4.1 O HotDraw

O framework HotDraw é um editor semântico[13] para gráficos estruturados, e pode ser usado para criar editores para gráficos bidimensionais especializados como diagramas esquemáticos, *flowcharts* e ferramentas CASE [Johnson,92].

O HotDraw foi desenhado em meados dos anos 80 por Kent Beck e Ward Cunningham quando estavam na Tektronix Inc. O HotDraw foi escrito originalmente em Smaltalk-80 e posteriormente reimplementado com novas características. Patrick

[13] editor semântico – permite associar conceitos e significados aos elementos gráficos tratados`..

McClaughry o atualizaram para o Smaltalk R.4.1 da VisualWorks. Sua manutenção na atualidade tem ficado a cargo Ralph Johnson e John Brant [Brant,01]. Uma versão comercial é mantida com implementação em Java chamada Drawlet [Rolemodelsoft,01].

A documentação existente sobre esse framework quase sempre não especifica a versão a que se refere. Esta abordagem se refere à versão 5.0 do HotDraw, procurando apresentar aspectos e conceitos que sejam comum ás versões existentes.

Tecnicamente, o Hotdraw é classificado como um framework *Gray-Box*, pois aplicações podem ser desenvolvidas por herança e por composição de classes concretas. A figura 4.1 apresenta um exemplo de interface gráfica do usuário básica para um editor baseado no HotDraw.

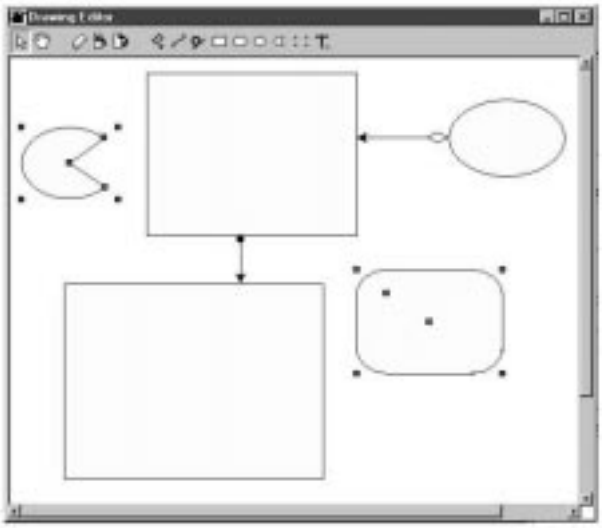


Figura 4.1: Interface gráfica típica de um editor baseado no HotDraw.

Estrutura de Classes do Hotdraw

As classes principais do framework HotDraw são: *Drawing*, *Figure*, *Tools* e *Handle*, *DrawingView* e *DawingController*. A classe *DrawingEditor* é a superclasse (concreta) de qualquer editor específico e sendo concreta, contém a implementação de um editor gráfico básico.

As figuras são os principais elementos de qualquer desenho. No conceito de domínio do HotDraw, um desenho (*Drawing*) é uma coleção de figuras(classe *Figure*).

Uma *Figure* está sempre associada a um componente visível do desenho, o *DrawingView*. Esta última mantém o modelo de dados da *Figure* em uma outra estrutura(*DrawingModel*).

Tool é a classe que representa a ferramenta ou interface do usuário capaz de alterar o *status* do desenho. *Handle* é a classe que representa elementos visuais (pontos) usados para controlar características da figura, tais como dimensões, por exemplo.

Um controlador (*DrawingController*) agrega as funcionalidades de ferramentas e desenhos.

O diagrama da figura 4.2 foi montado a partir dos trabalhos de Froehlich [Froehlich,01] e de Buhr [Buhr,99] num comparativo com o software original.

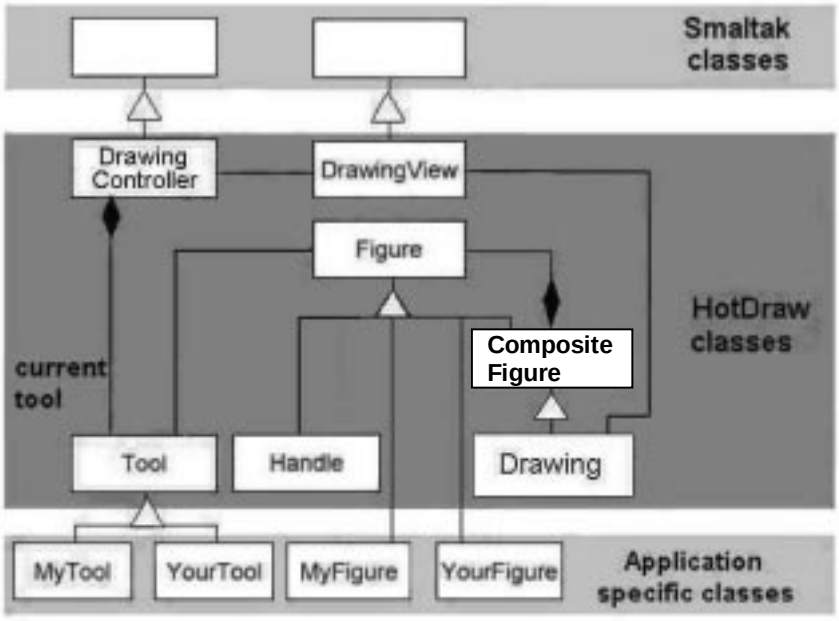


Figura 4.2: Classes relevantes do HotDraw

O HotDraw é um framework fortemente baseado na arquitetura MVC original do Smalltalk, e os dois aspectos mais fortes desta abstração no HotDraw são as figuras e as ferramentas de gerenciamento destas figuras

A classe *Figure* estende classes para primitivas gráficas a fim de permitir a abstração de figuras mais complexas através da classe *CompositeFigure*. Esta última identifica um padrão usual em frameworks deste domínio de aplicação e será discutida,

no capítulo 5. O mesmo ocorre também com a classe *Tools* que tem como herança ferramentas mais elaboradas e com funcionalidades distintas.

Figuras do HotDraw.

Originalmente o HotDraw implementa algumas classes para comportar as abstrações de figuras primitivas que podem ser usadas isolada ou conjuntamente e podem ser:

TextFigure	classe usada para gerenciamento de textos sobre o desenho;
ElipseFigure	classe usada para criar e manter figuras elípticas inclusive o círculo;
PolygonFigure	classe que abstrai figuras poligonais e que para este caso já estende a linha e o retângulo, classes *RectangleFigure* e *Line* respectivamente.
CompositeFigure	agrega *figures* permitindo a construção de figuras compostas.

A figura 4.3 apresenta o relacionamento entre essas classes [Buhr,99].

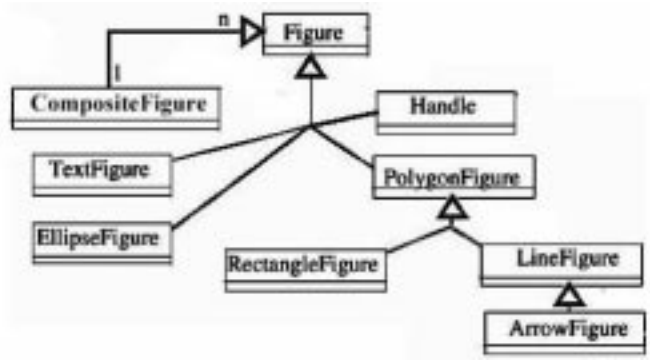


Figura 4.3: Classe *Figure* e suas subclasses.

Ferramentas do HotDraw

Um editor gráfico criado a partir do HotDraw possui um elenco de ferramentas que devem atuar tanto para criação de novas figuras como para manutenção das já existentes. Apesar de possuir diversas ferramentas, apenas uma pode esta ativa num determinado instante. Quando uma figura é selecionada pela ferramenta, ela apresenta

um conjunto de *handles*. Manipulando um *handle* algumas propriedades da figura são mudadas.

Observando a interface gráfica de uma aplicação HotDraw é possível constatar as principais classes de ferramentas providas pelo framework, conforme nos apresenta a figura 4.4.

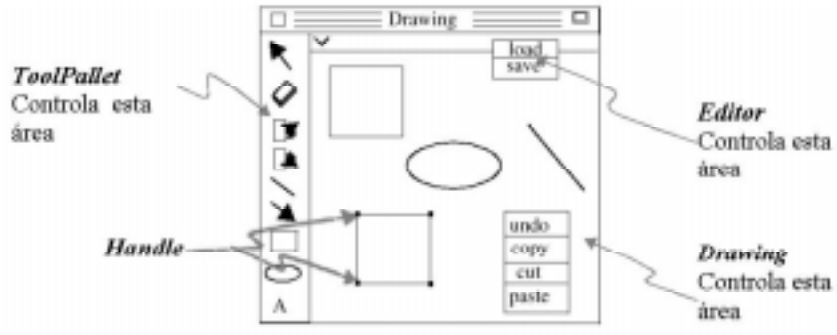


Figura 4.4: Ferramentas do HotDraw.

As principais ferramentas contidas em um editor padrão são representadas pelas classes:

SelectTool	classe que gerencia a seleção dentro da área de seleção;
EraseTool	classe que abstrai a ferramenta usada para remover figuras;
ShuffleUpTool	corresponde à enviar a figura para frente;
ShuffleDownTool	descreve a funcionalidade de enviar a figura para traz;
FigureCreationTool	esta parte descreve a funcionalidade relativa à ferramenta de criação de figuras.

A figura 4.5 apresenta a classe *Tools* e seu relacionamento com suas principais subclasses [Buhr,99].

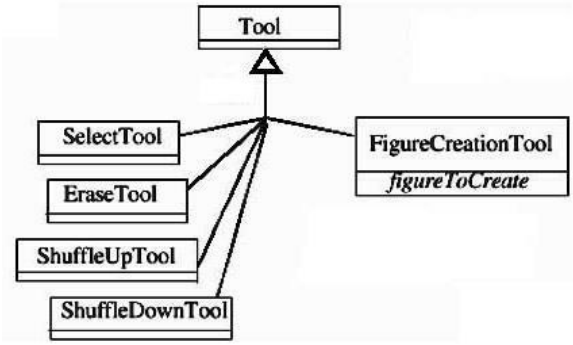


Figura 4.5: Classe *Tools* e suas subclasses.

Implementando editores com HotDraw

Uma aplicação para usar HotDraw necessariamente deve ser implementada em *Visual Works Smalltalk, o* que é um fator é limitante no desenvolvimento de aplicações comerciais, devido à baixa disseminação dessa linguagem.

É fundamental também conhecer os *hot spots* e *frozen spots* do projeto, pois são considerados na seqüência de implementação.

Na a figura 4.6, as classes que possuem círculos em seu interior são os *frozen spots* e as que possuem a extremidades chanfradas são os *hot spots* do framework [Froehlich,01].

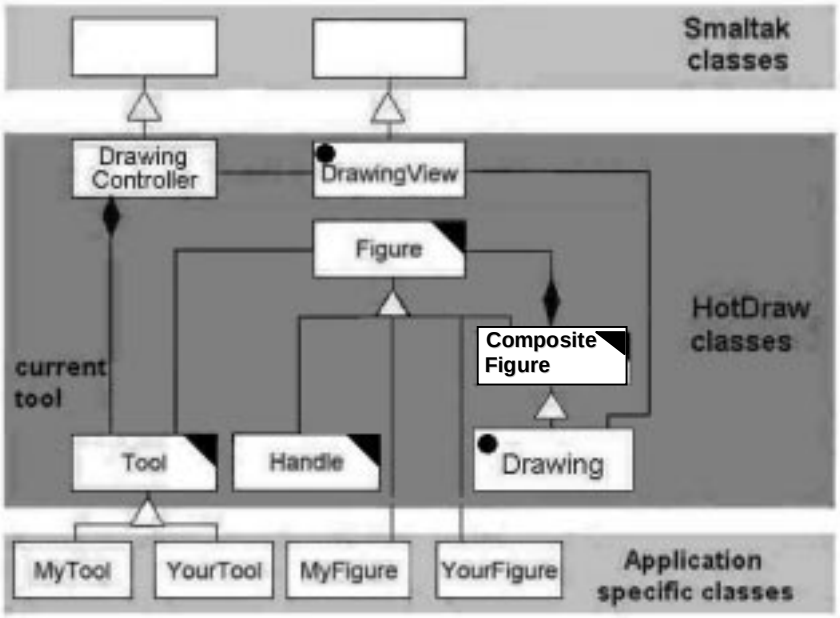


Figura 4.6: *Hot Spots* e *Frozen Spots* do Hotdraw.

A documentação para implementação de aplicações sob o HotDraw é escassa, o que existe disponível são exemplos. A prática mais comum tem sido estudar os exemplos que acompanham o pacote e entender a aplicação.

4.2 JHotDraw

O JHotDraw é também um editor semântico de gráficos implementado em linguagem Java, criado por Kent Beck e Erich Gamma [Gamma,95]. É considerado um

framework maduro e sua concepção é inspirada no HotDraw e sua motivação principal também foi a prática didática. A terminologia adotada para o HotDraw bem como a correlação entre figuras e desenhos são usados aqui com o mesmo significado [Riehle,00][Brant,01].

Esse framework difere do HotDraw original não especificamente pela linguagem de implementação, mas pela forte orientação à padrões. E considerado também como um caixa-cinza e, ao todo consegue implementar 8 padrões bem definidos: *Observer, Composite, Strategy, State, Template Method, Decorator, Factory Method e Prototype.*

O JHotDraw é fortemente baseado em padrões, pois na concepção do seu criador "nenhuma linguagem pode reduzir a importância do *design*" [Gamma,95].

As particularidades da linguagem Java em relação à orientação a objeto na implementação desse padrão serão apresentadas no Anexo I.

Estrutura de Classes do JHotDraw

A figura 4.7 apresenta um diagrama com as classes mais relevantes.

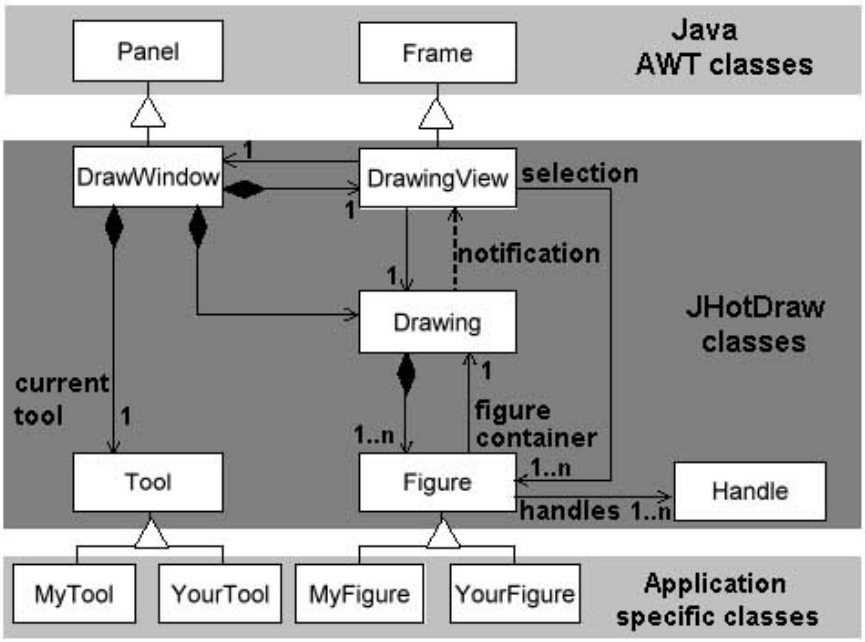


Figura 4.7: Classes relevantes do JHotDraw[Gamma,95].

Neste caso, os *hots spots* estão melhor definidos, e constituem-se das classes que interface com as classe da tarja *Application specific classes*. O comentário sobre esse

framework será feito a seguir, usando a notação UML e a descrição das subclasses se refere às principais classes de cada diagrama.

A figura 4.8 faz uma apresentação sintética das classes e subclasses básicas deste framework. As classes em cinza representam os *hot spots* do framework.

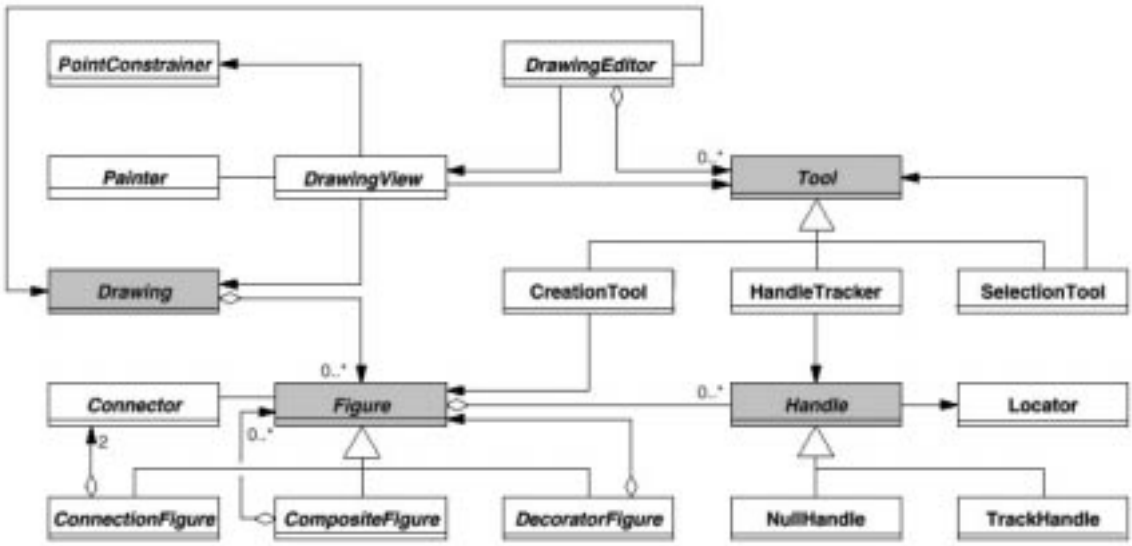


Figura 4.8: Classes e subclasses principais do JHotDraw [Riehle,00].

Classes *Figure*

A classe *Figure* é composta pelas seguintes subclasses:

Figure	Todo e qualquer elemento que pode ser representado na GUI. Uma figura pode ser composta por diversas figuras. Toda figura possui um grupo de *handles* para gerenciar suas características.
CompositeFigure	é a figura composta por diversas outras.
DecoratorFigure.	diz respeito aos acessórios da figura, tal como a borda;
ConnectionFigure.	descreve uma figura que conecta duas outras figuras;
Connector	descreve o ponto de conexão entre figuras;
Handle	pontos de controle para gerenciamento de propriedades da figura;
Drawing	é um *container* de figuras;

FigureChangeListener é o objeto que monitora as mudanças na figura;
DrawingChangeListener objeto para monitoramento de área de desenho.

Classes Drawing e DrawingView

As classes *Drawing* e *DrawingView* apresentam uma estrutura formada pelas seguintes subclasses :

DrawingView cria o desenho e inspeciona sua alteração;
Painter encapsula um algoritmo para pintura de regiões da figura;
PointConstrainer força o alinhamento de um ponto do desenho à uma posição na grade;
Tool refere-se às ferramentas e suas respectivas funcionalidades para gerenciar desenhos.

Classes DrawingEditor

As classes relacionadas à classe *DrawingEditor* são:

EditorMediator é um gerenciador de comportamento dos objetos de desenho e permite desacoplá-lo quando necessário;
ToolAccess controla requisições do cliente da ferramenta em uso (corrente);
ToolCreation gerencia a criação de objetos do tipo *Tool*;
ToolState é um modelo do comportamento do *handle* manipulado de um determinado objeto de desenho.

Organização do Pacote

Todas as classes e interfaces do JHotDraw estão organizadas em 8 pacotes agrupados de acordo com suas funcionalidades.

A implementação de uma aplicação depende da compreensão desta organização que inclui até exemplos.

A tabela 4.1 apresenta um descritivo desta organização, especificando o conteúdo de cada pacote.

Pacote CH.ifa.draw.	Conteúdo
Util	utilitários em geral e podem ser usados sem o JHotDraw
Framework	classes e interfaces que definem o framework sem implementação.
Standard	Implementações padrão para as classes do pacote anterior.
Figures	um kit de figuras e ferramentas (tools, handle) disponíveis
Contrib	classes contribuintes de outras.
Applet	interface para implementação de applets.
Application	interface para desenvolvimento de aplicação standalone.
Samples.*	aplicações exemplo desenvolvida a partir do JHotDraw.

Tabela 4.1: Organização do pacote do JHotDraw

Implementando com JHotDraw

A tabela 4.2 apresenta os níveis de reuso proporcionados por uma aplicação com base no JHotDraw. É importante observar que classes cujo nome se inicia com "*standard*" indica que não necessitam ser implementadas, já as iniciadas com "*abstract*" sim.

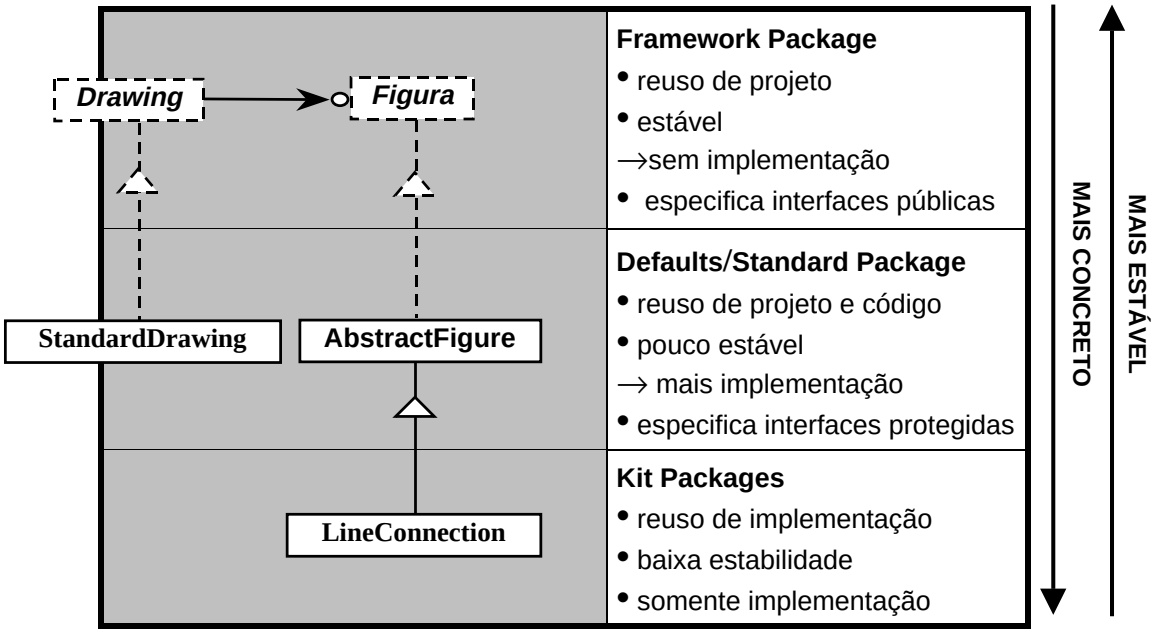


Tabela 4.2: Nivelamento do Framework.

Os padrões descrevem a estrutura do framework, em forma hierárquica, permitindo entender cada uma de suas abstrações de forma fragmentada.

Este aspecto torna-se um agravante quando a aplicação apresenta um grande número deles, dificultando ao desenvolvedor a formulação de uma visão de conjunto da aplicação.

4.3 Outros frameworks gráficos

Os dois frameworks anteriores foram selecionados considerando a identidade da abstração empregada. Outros frameworks com estruturas similares ou distintas forma projetados com destaque para o Unidraw [Vlissides,90].

Unidraw

O Unidraw é um framework para gráficos estruturados criado por Vlissides [Vlissides,90] projetado sobre a biblioteca InterView. O êxito do Unidraw é creditado a sua produtividade, pois permite acelerar o processo de implementação de aplicações.

A figura 4.9 apresenta a estrutura de camadas em que se baseia o Unidraw. É importante observar que uma aplicação sob Unidraw reusa tanto classes do Unidraw como também da biblioteca que lhe dão suporte, o InterView [Vlissides,90].

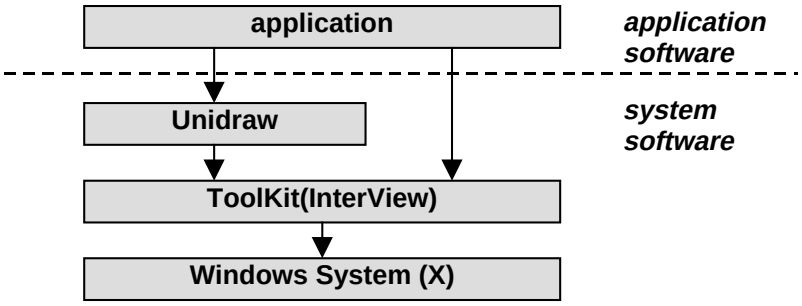


Figura 4.9: Camadas de uma aplicação sob Unidraw [Vlissides,90]..

Define basicamente quatro abstrações: *components* que encapsula a aparência e o comportamento dos objetos, *tool* suporte direto à manipulação dos componentes, *command* que define as operações sobre os componentes; e *external representations* que define o mapeamento entre os componentes e os dados referentes ao objeto .

Agrupamentos

Segundo Johnson [Johnson,02], o Hotdraw e o Unidraw não são efetivamente os frameworks gráficos mais estáveis, e talvez nem os melhores, mas pelo fato de serem de domínio público e de forte veiculação acadêmica, acabaram por se tornar uma forte referência para estudo de frameworks desse domínio de aplicação.

O projeto de framework a partir de outro do mesmo domínio, cuja abstração já esteja consolidada é uma pratica bastante usual. Na grande maioria das vezes, a idéia é simplesmente portá-lo para outra linguagem de programação, adaptá-lo, ou ainda implementar um recurso ou suporte adicional.

Uma varredura nos principais catálogos de frameworks gráficos de domínio público [SourceForge,02] [Tai,02] [Cetus,02] forneceu subsídios para a montagem da tabela 4.3. Os frameworks gráficos foram organizados hierarquicamente em três agrupamentos, de acordo com os frameworks em que se basearam (cinza).

Framework	Linguagem	Características importante	Hosting
HotDraw	**SmallTalk**	**Pequeno, maturidade**	**st-www.cs.uiuc.edu/users/brant/HotDraw/HotDraw.html**
CoolDraw	Smalltalk	Trabalho com constraint	www.ursamajor.uvic.ca
DrawLet	Java		www.rolemodelsoft.com/drawlets
GEF	Java		www.ics.uci.edu/pub/arch/gef
JHotDraw	Java	Uso intensivo de padrões	www.jhot.draw.org
Jkit/GO	Java	Suporte CORBA	www.objectspace.com/
MacDraw	C++		www.mac512.com/macdraw.htm
Mica	Java		www.swfm.com/mica
Unidraw	**C++**	**Qualidade, produtividade**	www.ivtools.org
Belim	Java	Servidor CORBA-based scene-graph, X-Window	Sourceforge.net/projects /berlin
Fresco Jfresco	C++ Java	unifica grafico/widget, suporte CORBA	www2.berlin-consortium.org/wiki /html/Berlin/TheFrescoProject.htm
Ilog	Java		www.ilog.com/products/jviews /graphframe/
Interview	C++	Biblioteca núcleo do Unidraw	www.ivtools.org/ivtools /interviews.html
Ivtool	C++	Multi-frame, muiti-layer	www.ivtools.org/ivtools
Outros			
GFST	Smalltalk		www.smalltalksystems.com /gfst.htm
Pingo	Python		Pingo.netpedia.net/
Sketh	Python		http://sourceforge.net/projects /sketch

Tabela 4.3: Quadro de frameworks gráficos [SourceForge,02] [Tai,02] [Cetus,02].

4.4 Comparando Abstrações

Como é possível observar na tabela 4.3, no domínio de editores gráficos dois frameworks tiveram larga aplicação e se tornaram conhecidos: o Hotdraw e o Unidraw. Este fato nos permitiu agrupar os demais de acordo com framework de origem, diferindo apenas em aspectos secundários mas mantendo o escopo da abstração. O entendimento deste aspecto é fundamental para propor novos frameworks que agreguem aspectos evolutivos.

Grupo HotDraw x Grupo Unidraw

Como se pode constatar, o tratamento dado à gestão das figuras é diferente nos dois grupos, ou seja, o mesmo domínio de aplicação é implementado com estratégias distintas.

Segundo Johnson [Johnson,02], apesar de terem sido desenvolvidos separadamente, possuem muitas semelhanças.

A tabela 4.4 apresenta um quadro comparativo de abstrações básicas e as classes que implementam cada uma delas.

<table>
<tr><th colspan="4">ABSTRAÇÕES</th></tr>
<tr><th colspan="2">HotDraw</th><th colspan="2">Unidraw</th></tr>
<tr><td>Editor</td><td>DrawingController</td><td rowspan="2">External representation</td><td rowspan="2">Editor, Component</td></tr>
<tr><td>Drawing</td><td>Drawing, Figure</td></tr>
<tr><td>View</td><td>DrawingView</td><td>Components</td><td>Viewer, ComponentView</td></tr>
<tr><td>Tool</td><td>Tool, Handle</td><td>Tool, Command</td><td>Tool, Catalog</td></tr>
</table>

Tabela 4.4: Classes correlacionadas as abstrações.

Observa-se que a quantidade de classes que tem responsabilidade sobre uma abstração básica varia em cada grupo. Um exame na documentação do Unidraw [Vlissides,90] esclarece que diferem essencialmente na concepção das ferramentas de gerência da área de desenho e na estrutura das figuras.

Outra diferença notável é a de que o Unidraw possui mais implementação, o que o aproxima bastante de um framework tipo caixa-branca, enquanto que o HotDraw é um genuíno caixa-preta. [Brant,01] e é mais protótipo, o que o torna mais apropriado aos objetivos deste trabalho.

Grupo HotDraw: HotDraw e JHotDraw

Uma comparação entre as figuras 4.2 e 4.7 aponta para semelhança na abstração empregada. Além da linguagem de implementação o relacionamento entre as classes é o fator mais significativo.

Observa-se que esses frameworks apresentam certas similaridades, mesmo usando semântica diferente. Geralmente apresenta uma espécie de *kernel*, uma estrutura central composta pelas classes relevantes que definem a solução. Na borda desta estrutura, encontram-se geralmente os padrões e soluções de implementação que constituem os *hot spots* do framework.

Um ponto considerado divergente entre as abstrações é o relacionamento entre figuras e os desenhos propriamente ditos(*figure-drawing*), evidenciando apenas uma diferença conceitual, provavelmente porque o JHotDraw já se preocupa mais em incorporar também as abstrações já absorvidas pela linguagem Java[14].

Outro aspecto a considerar é o fato do Unidraw ser projetado sobre uma biblioteca de classes pré-existente[Vlissides,90], o que certamente impõe limitantes ao seu projeto.

4.5 Arquitetura MVC

O MVC é um framework muito utilizado na modelagem de sistemas interativos baseados em GUI. Foi concebido inicialmente para desenvolvimento para o Smalltak em 1970 pela equipe da Xerox PARC. O estudo do MVC contribuiu para a concepção

[14] classes Java de figuras (rectangle, line, etc) e ambientes de desenho como JFrame,JDesktop, JPanel, etc.

dos frameworks [Goldberg,83]. Para se ter uma dimensão de sua importância, a quase totalidade dos frameworks gráficos o utilizam.

A abstração básica do MVC pressupõe uma separação entre a interface gráfica e os dados que ela apresenta.

O MVC possui três componentes [Buchmann,96]:

Model **-** encapsula dados manipulados e o núcleo das funcionalidades da aplicação. Ele é independente das representações específicas apresentadas na saída;

View **-** mostra informações ao usuário a partir de dados obtidos do *Model*. Podem existir múltiplas visualizações do modelo;

Controller - recebe eventos da entrada e traduz em serviços para o *Model* ou para o View. O usuário só interage com a aplicação através deste componente.

Na maioria das linguagens de programação, a função do *controller* é desempenhada pelo próprio sistema geral de notificação de eventos.

O diagrama da figura 4.10 procura vincular os periféricos de um PC convencional com um dos componentes do MVC.

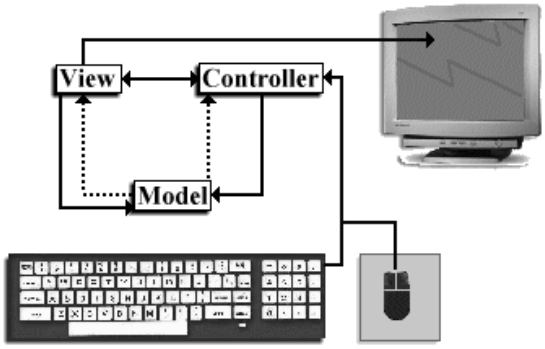


Figura 4.10: Componentes do MVC.

Vantagens do MVC

O uso deste framework permite desvincular a aplicação dos dados permitindo assim múltiplas visões para um mesmo modelo.

Linguagens como o *Smalltalk* permitem a sua utilização direta como um padrão, outras provêem padrões como o *Observer* a fim de permitir sua implementação, como é o caso do Java.

Diversas combinações de encapsulamento podem ser feitas entre os três componentes, podendo adequá-la à forma de utilização da aplicação. O encapsulamento do *view* com o *controller* gerou o conceito de *delegate*(ou MV), uma conceito muito próximo das GUI´s proporcionadas pelas linguagens como o Java.

Os detalhes da sua implementação desses recursos em Java estão no Anexo I.

O paradigma MVC pressupõe uma mudança de estado do sistema vinculada às alterações no estado da interface gráfica do usuário. As mudanças que ocorrem no sistema devem refletir imediatamente na GUI.

A flexibilidade desse framework permite a construção de GUI´s, com estrutura de projeto adaptável dinamicamente e permite o reuso de suas funcionalidades. É importante observar que o grau de padronização das interfaces gráficas é um fator determinante para o bom aproveitamento desta arquitetura.

O próximo capítulo discutirá a proposta de desenvolvimento de um framework para construção de aplicações gráficas que possua semântica associada, implementado em Java.

5. O JHOTSEA

A diversidade de frameworks gráficos descritos no capítulo 4 permite a identificação de aspectos do domínio que dificultam sua modelagem. A obtenção de modelos de domínio abrangentes e a necessidade de adequação aos novos recursos de implementação são algumas das dificuldades comumente encontradas.

A proposta do framework deste trabalho, o JHotSea, procura herdar conceitos do grupo do HotDraw, permitido assim o "reuso" de aprendizado dos usuários daquele framework.

O JHotSea é um framework voltado para o desenvolvimento de componentes de edição gráfica a ser incorporado em aplicações mais complexas, tal como o ambiente de desenvolvimento de software Rational Rose [Rational,00] e o SEA [Silva,00].

O JHotSea procura também incorporar recursos avançados de Java 2[15][Deitel,00], tais como os disponibilizados pelos pacotes Java.net que permitem atuação em ambiente de internet, o Swing, o pacote Collections, etc.

Assim, este capítulo trata da modelagem e implementação JHotSea, discutindo e justificando as soluções adotadas, encerrando com a apresentação de dois exemplos de aplicação desenvolvidas a partir do JHotSea.

---

[15] Java 2 – Tecnologia desenvolvida a partir da versão 1.2 da linguagem Java.

5.1 Modelagem Conceitual

A definição do domínio da aplicação exige uma especificação clara dos conceitos a serem utilizados. A seguir são descritos os conceitos mais relevantes envolvido no projeto do JHotSea.

Gap semântico[16]

Uma das dificuldades no uso de frameworks está em aprender a utilizá-lo, pois o desenvolvedor necessita dominar os três níveis de abstração: as abstrações da linguagem de implementação (classes, padrões e frameworks), a abstração do projeto do framework daquele domínio da aplicação para poder usá-lo e, a compreensão da própria aplicação com suas interfaces gráficas, funcionalidades, etc.

Uma forma de minimizar esta distância ou *gap* semântico seria aproximar os conceitos usados nas abstrações. No caso, duas aproximações são possíveis: entre a linguagem e o projeto, e entre a aplicação e o projeto. No primeiro caso, como pudemos constatar no capítulo anterior, a própria linguagem Java já incorporou muitas das abstrações relativas à figuras e interfaces gráficas, podendo ser vista inclusive como uma linguagem orientada a framework. No segundo caso, um refinamento sistemático no projeto pode reduzir esse *gap* semântico.

View, Model e Constraint

A diferenciação entre elementos *View, Model e Constraint* é fundamental para a compreensão do framework.

Elementos *View* definem operações que apresentam o desenho na interface gráfica, ou seja, definem como desenhar a figura.

Elementos *Model* definem propriedades e características de uma figura. Um objeto *View* utiliza dados do objeto *Model* dessa mesma figura para desenhá-la. Toda alteração na figura (mover, editar, apagar,etc) é realizada no *Model* para em seguida ser

---

[16] Esse termo normalmente é usado em outro contexto para expressar distâncias na representação semântica de conceitos, aqui o sentido original é reusado para o caso de abstrações.

apresentada pelo *View*. Para cada objeto *Model* existe necessariamente um *View* correspondente.

Elementos *Constraint* definem restrições de comportamento de uma figura. Para esse trabalho, optou-se por associa-las ao *Model*. Desta foram, a criação ou qualquer alteração em figuras causariam mudanças no modelo da figura, e este modelo fica sujeito as restrições do *constraint*. São elementos de implementação opcional.

Outros conceitos podem ser incluídos de acordo com a necessidade. Para a geração de código vinculado a figuras, por exemplo, bastaria criar um conceito *Code*, e associá-lo ao *Model*.

Portanto, uma figura qualquer possui classes que representam o conceito *View* (*figureView*), conceito *Model* (*figureModel*) e opcionalmente o *Constraint (figureConstraint*). A figura 5.1 ilustra as classes que representam esses três conceitos referentes a uma figura tipo linha (*line*).

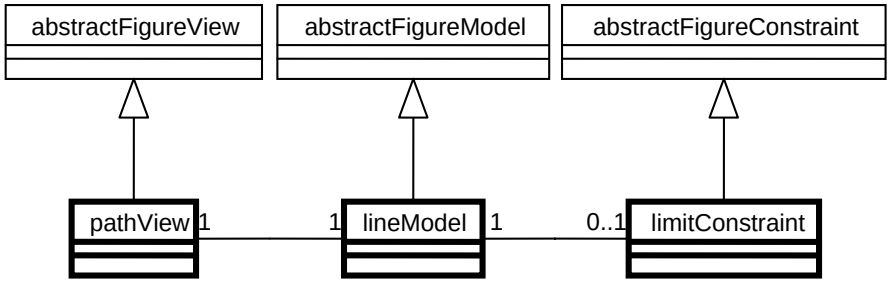


Figura 5.1: Classes representando os conceitos *View*, *Model* e *Constraint* de um *line.*

Observe que o *constraint limitConstraint* e o *view pathView* podem estar também associados a outros *figures*.

Qualquer outro conceito a ser acrescentado deveria ser associado à classe *lineModel*.

A figura 5.2 apresenta classes que mostram a relação destes conceitos. Observe nela a introdução de novos *figuresView* (*rectangleView* e *lineView*) e de *figureModels* correspondentes (*rectangleModel* e *lineModel*).

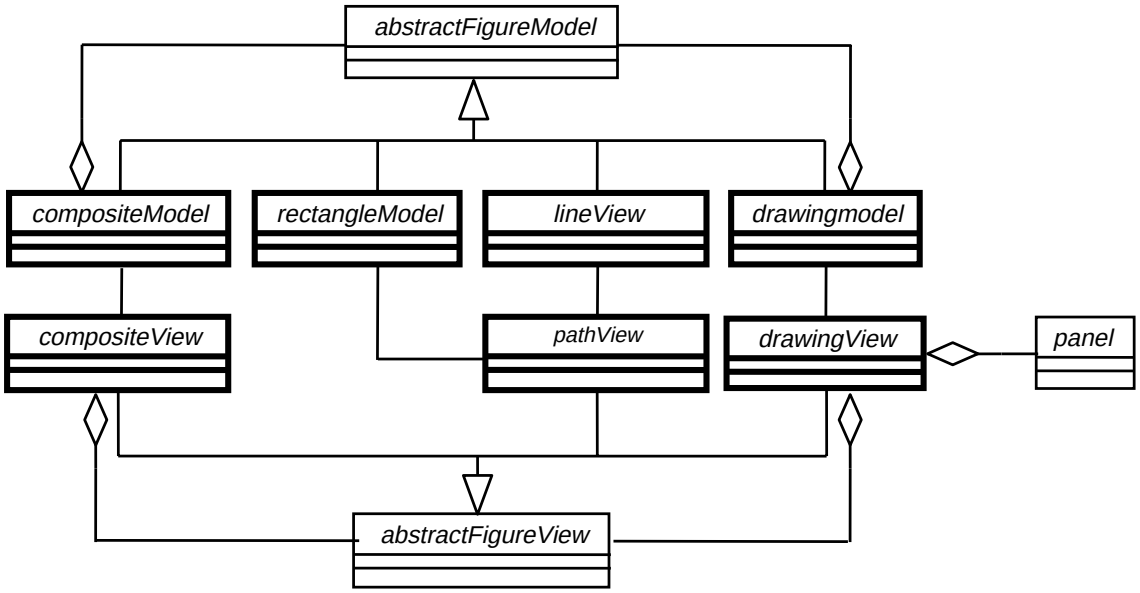


Figura 5.2: Diagrama de Classes Model e View.

5.2 Análise de Domínio

Toda a compreensão do domínio de aplicações voltadas para a edição de gráficos estruturados parte da analise das ferramentas usuais, dos seus elementos e conceitos. A abordagem adotada busca descrever os elementos que compõem uma aplicação genérica deste domínio identificando a relação entre esses conceitos.

A tarefa então fica restrita a identificar os objetos conceituais que representam esses elementos, isso pode ser feitos selecionando objetos (classes) relevantes desse contexto.

Classes relevantes

Um framework gráfico generaliza características reusáveis em várias aplicações voltadas para a edição gráfica. A estrutura de classes do JHotSea contém parte das abstrações capturadas pelo Hotdraw e JHotDraw, preservando as classes centrais tais como *Figure*, *CompositeFigure*, DrawingControler, *DrawingView e Tool*.

As classes *Handle* especificadas no Hotdraw e no JhotDraw gerenciam ferramentas de edição acopladas a própria figura. Esse recurso não é de fácil intuição por parte do usuário, razão pela qual foi suprimido. Assim, em JHotSea *a abstração Handle* foi suprimida e suas responsabilidades transferidas para outras classes.

A figura 5.3 apresenta as classes principais do framework JHotSea, relacionando-as com os elementos da interface gráfica do usuário, típica de aplicações deste domínio.

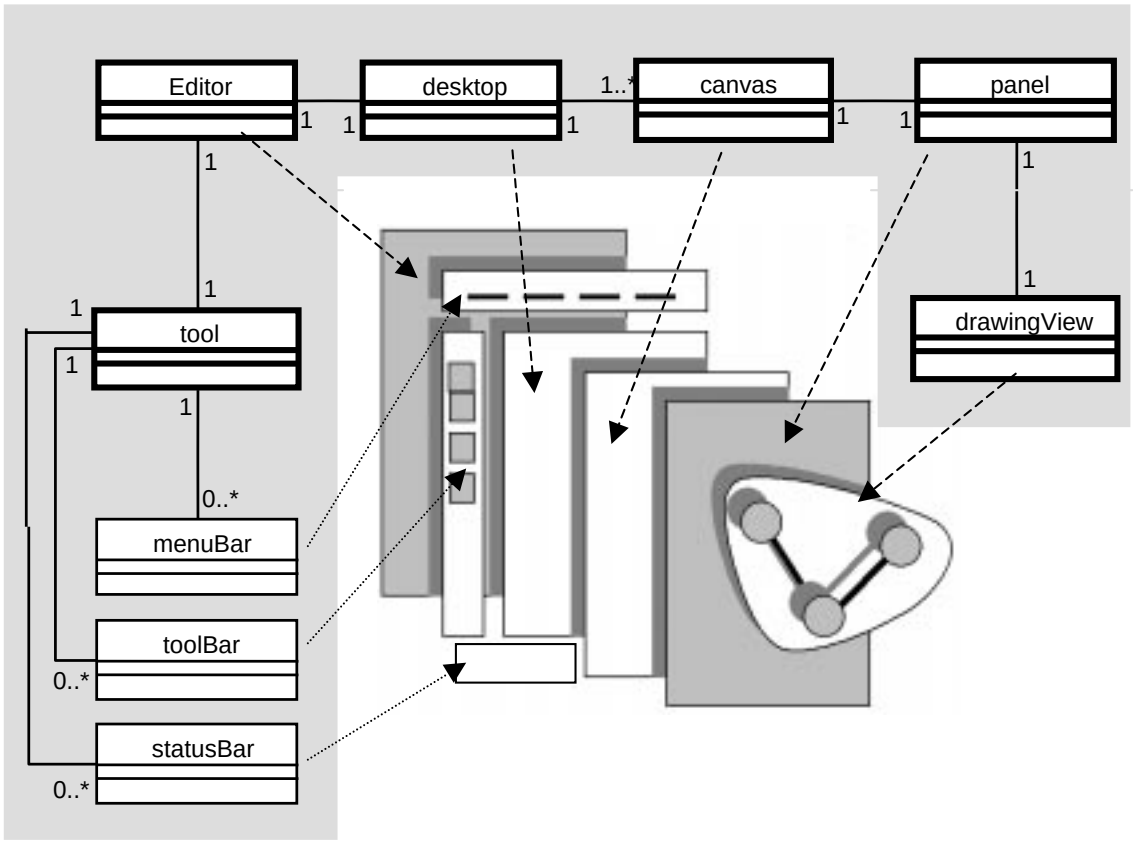


Figura 5.3: Classes Principais do JHotSea e os componentes de um editor gráfico.

Observe que para cada elemento da interface gráfica existe uma classe definida. O mesmo ocorre em relação à linguagem de implementação, pois cada classe é estendida a partir de uma classe Java específica para aquela função.

editor representa o editor propriamente dito e agrega os elementos gráficos de apresentação com as ferramentas disponíveis para manipulá-las;

desktop representa um *container* para criação de diversas janelas para desenho, confere suporte para MDI[17] ao framework;

[17] MDI – Capacidade da aplicação de abrir múltiplas janelas no mesmo contexto.

canvas é a janela de desenho propriamente dita, onde o gráfico será colocado. Um *desktop* pode conter diversos *canvas*;

panel um desenho não é colocado diretamente sobre o *canvas*. Primeiramente deve ser colocado sobre um painel (*panel*). Esta classe foi aqui inserida a fim de adequar às restrições das classes Java correspondentes[18];

drawingView trata-se da classe que contém as figuras, ou mais especificamente, objetos *View*. Tais objetos representam os elementos *View* do paradigma MVC, que definem como uma figura é desenhada no *panel*;

tool representa todo o ferramental disponível para manuseio de figuras. Tecnicamente, o Java disponibiliza para tal função as barras de ferramentas e de menus. Embora a linguagem permita agregar uma barra de menu diretamente ao editor, haveria prejuízo conceitual para a abstração, razão pela qual a mantivemos associada a *tool;*

menuBar representa a agregação de todas as barras de menu do domínio;

toolBar representa a associação de todas as barras de ferramentas do domínio;

Especificação de Documentos

Um framework se destina a gerar diferentes aplicações para um domínio. Logo, precisa conter uma descrição dos conceitos desse domínio [Silva,00]. Assim, a introdução de estruturas que possam conter essas descrições contribui não só no aspecto documental em si, mas pode ser instanciado para dar suporte em funcionalidades, como por exemplo a geração de códigos a partir de conceitos.

Essa abordagem está presente em trabalhos que versam sobre documentos estruturados e hiperdocumentos e tem forte formalismo ontológico [Guarinco,68].

---

[18] A segmentação da região do JFrame só pode ser realizada pela criação de instâncias JPanel's .

A adoção de repositórios na construção de ambiente é defendida por Martin a fim de manter a coerência e integridade das informações do ambiente e seus elementos [Martin, 95].

Um conceito é entendido como a unidade de informação do domínio e modelagem tratados e um modelo conceitual agrega um conjunto de conceitos usados em um determinado modelo (diagramas, código, etc.) [Silva,00].

Uma especificação agrega elementos de especificação que podem ser conceitos ou modelos, que podem estar ou não relacionados. Produzir uma estrutura consiste em definir um conjunto de tipos de modelos e de conceitos, e da forma como podem se relacionar.

A solução adotada baseia-se na proposta adotada pelo framework OCEAN[19], que trabalha com uma diversidade de conceitos gráficos e textuais.

A classe *specification* atua como um repositório de especificações de conceitos. A classe *typeObject* representa o tipo de alguns conceitos.

A figura 5.4 mostra o diagrama de classes com as classes envolvidas.

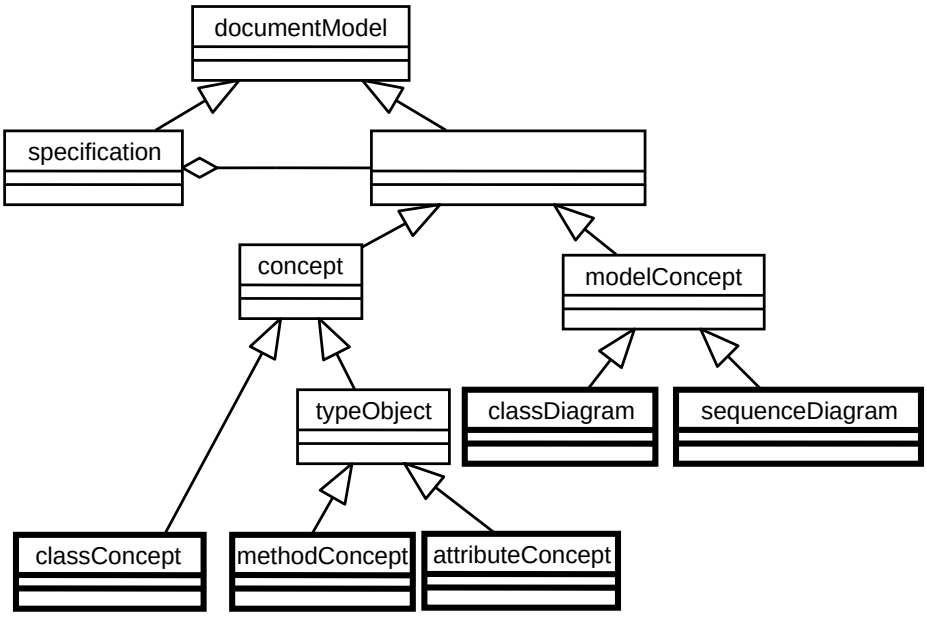


Figura 5.4: Diagrama de Classes: Modelos e Conceitos.

[19] OCEAN framework sob o qual foi desenvolvido o SEA, que é um ambiente CASE desenvolvido por Silva, objeto de tese de doutorado [Silva,00].

No diagrama da figura 5.4 foram criadas as subclasses para os modelos *classDiagram* e *sequenceDiagram* e três conceitos (*classConcept, methodConcept* e *attributeConcept*) os conceitos usados para modelos do tipo diagrama de classe.

Uso de Padrões

O JHotSea faz uso intensivo de padrões para sua implementação. Padrões como o *Abstract Factory* são empregados com freqüência no desenho de ferramentas (*abstractMenuBar*, *abstractToolBar* e *abstractStatusBar*) e figuras (figureV*iew*, *figureModel* e *figureConstraint*), conforme é apresentado pela figura 5.5.

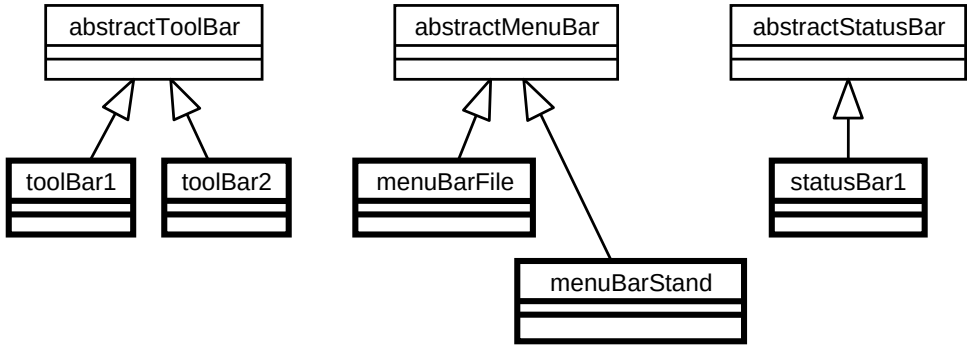


Figura 5.5 : Padrão *Abstract Factory*.

Na figura 5.2 é possível observar o padrão *composite*. A linguagem Java já disponibiliza classes e interfaces para implementação de padrões. O padrão *Adapter* usado para modificar interfaces é disponível pelas classes Java *WindowAdapter* e *MouseMotionAdapter*, por exemplo.

Como já foi exposto no capítulo anterior o uso de delegação de eventos contempla o padrão *Observer*. As interfaces Java *mouseEvent*, *mouseMotionEvent*, *windowEvent* e *keyEvent* implementam esse padrão.

5.3 Modelo de Figuras

A modelagem de figura segue três padrões: unidimensionais, bidimensionais e para o caso específico do texto.

Um modelo genérico pode ser obtido a partir de um retângulo e nas figuras que podem ser inscritas nele [Milne,02]. Assim, um modelo pode ser representado pela chamada da função (5.1):

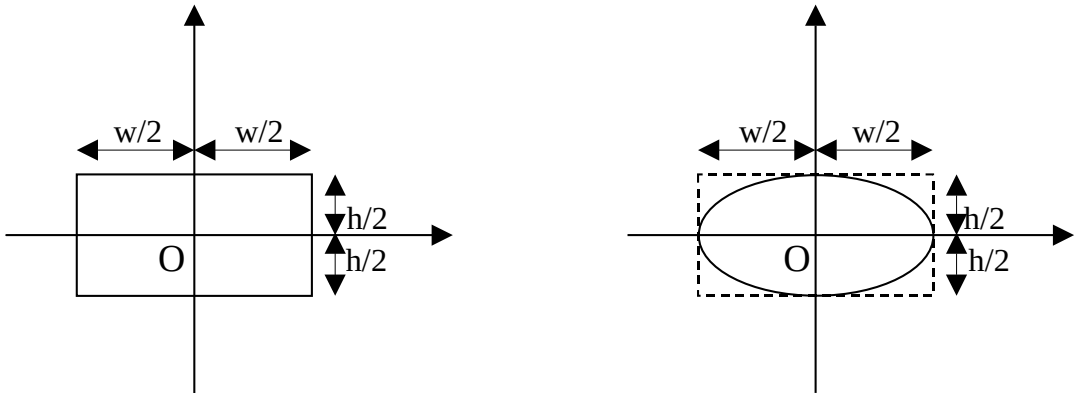
figureModel=object(Ox,Oy,width, heigth, string); *(5.1)*

Onde *Ox* e *Oy* correspondem às coordenadas do ponto de origem, *width* e *heigth* correspondem a largura e altura de um retângulo, e *string* à uma sequência de caracteres que se pode escrever dentro do retângulo.

O conjunto de operações disponíveis para cada uma delas varia de acordo com as características a seguir.

Figuras Bidimensionais

A figuras 5.6 apresenta o retângulo e a elipse e o método construtor Java correspondente.



fm=new rectangle2D.Double(Ox,Oy,w,h); fm=new line2D.Double(Ox,Oy,w,h);

Figura 5.6: Modelo de retângulo e círculo.

Para definir figuras primitivas mais elaboradas é necessário definir o caminhamento (*path*) para ser desenhado.

A figura 5.7 apresenta o triângulo e o losango inscritos em um retângulo:

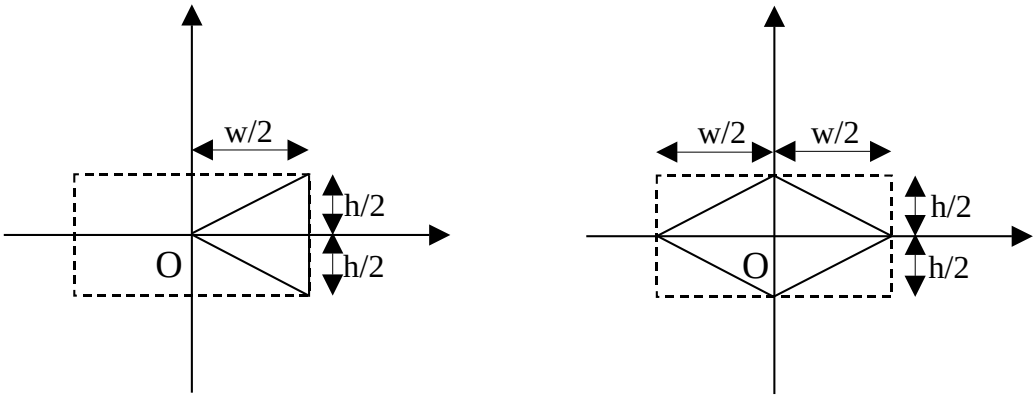


Figura 5.7: Modelo de triângulo e losango.

A implementação de modelos tipo *path* em Java usa a classe *GeneralPath* [Deitel,00]. O trecho de código 5.2 para implementação de o triângulo da figura 5.7(a) pode ser:

```
GeneralPath triangle2D=new GeneralPath();      // Cria o modelo
triangle2D.moveTo(Ox,Oy);                      // Fica a origem da figura
triangle2D.lineTo(w/2,h/2);                    // Desenha linha até um ponto
triangle2D.lineTo(w/2,-h/2);
triangle2D.closePath();                       // Fecha a figura ligando à origem
...
```

(5.2)

Transformação de figuras

Em tese, qualquer figura poderia ser construída a partir de uma primitiva semelhante com dimensões fixas, desde que se possa aplicar uma escala em cada eixo cartesiano. Para posicionar uma figura em do espaço basta aplicar-lhe um movimento de rotação e um de translação

A classes Java *AffineTransform* representa a notação matricial aplicável ao contexto gráfico para alterar e reposicionar uma figura ou conjunto delas. Esse procedimento é usual em computação gráfica e pode ser usado de forma transparente neste caso, ocultando assim a complexidade das operações matriciais implicadas nessas operações[Milne,02].

O trecho de código 5.3 abaixo reposiciona a figura mudando a sua origem , inclinação e reduzindo suas dimensões no eixo Y pela metade.

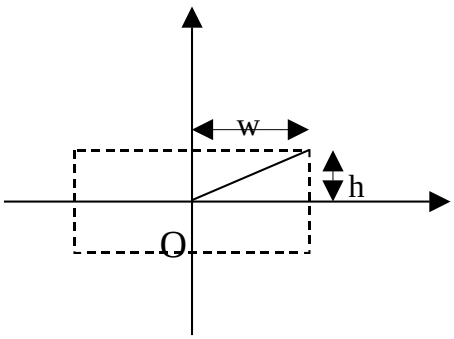
```
...
AffineTransform CA=new AffineTransform ();     // Cria contexto gráfico
CA.getTransform(g2D);                          // Captura o contexto corrente
AffineTransform AT1=new AffineTransform ();    // Cria contexto gráfico
AT1. translate( 50.0, 50,0);                   // reposiciona a origem da figura
AffineTransform AT2=new AffineTransform ();
AT2. rotate(Math.toRadians(theta));            // Inclina o eixo da figura em
AffineTransform AT3=new AffineTransform ();    // graus
AT3. scale(1.0,0.5);                           // Altera a escala no eixo Y
AT1.preConcatenate(AT2);                       // Concatena as transformações
AT1.preConcatenate(AT3);
AT1.preConcatenate(CA);
g2D.setTransform(AT1);                         // Aplica o contexto na área
...                                            // gráfica
g2D.draw(triangle2D);                          // Desenha a figura
...                                                                  (5.3)
```

Outro aspecto importante é que o próprio desenho (*drawingView*) pode ser Ter seu contexto sujeito a transformação permitindo assim recursos de *zoom in* e *out.*

Caso a figura deva estar conectada a outra, o modelo desta ultima deve estar associado a primeira por meio de um atributo.

Figuras Unidimensionais

A definição dos modelos dessas figuras inclui o ponto de inserção e os deslocamentos na horizontal e vertical do segundo ponto. Instanciam um objeto Java *Line2D,* conforme nos mostra a figura 5.8.



fm=new Line2D.Double(Ox.Ov.w.h);

Figura 5.8: Modelo da figura tipo linha

Aqui é possível também usar a *GeneralPath* para criar linhas quebradas, curvas e etc.

Texto

Uma cadeia de caracteres também prescinde de regras para a sua disposição na área gráfica. Para o caso os parâmetros *width* e *height* são dispensáveis, pois na verdade as dimensões finais do texto na área gráfica é que vão determinar as dimensões do retângulo em que ele está inscrito.

Assim, para o JHotSea qualquer figura pode ser a composição de três tipos de *view*'s: textos (*textView*), caminhos(*pathView*) e figuras puras(*figureView*).

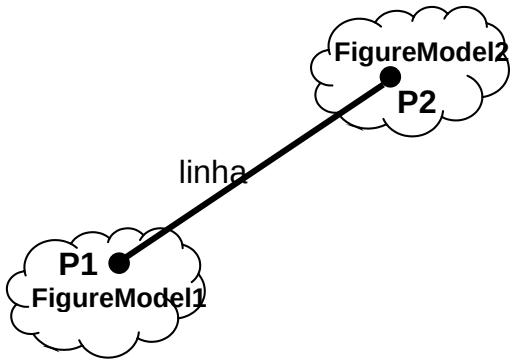
Composite

A associação de figuras na formação de elementos gráficos complexos tem solução simplificada, instanciando-se a primitivas dentro de subclasse de *compositeFigureModel*. Toda transformação de contexto aplicada a subclasse é imediatamente repassada ás primitivas. Assim, o trecho de código do construtor da classe *classFigureModel* com dois retângulos e um texto poderia ser (5.3):

```
public classFigureModel(){
        rfm1=new rectangleFigureModel(0,0,40,20);
        rfm2=new rectangleFigureModel(0,0,40,5);
        tfm=new textFigureModel(0,0,0,0,"class");
        ...
}
```
(5.3)

Conexão entre figuras

A conexão entre figuras é efetivada permitindo que um modelo, ao invés de pontos de inserção fixe a referencia de modelos da figura com que vai se conectar, como está representado na figura 5.9. Assim, uma linha para se conectar com dois retângulos assinala o modelo dessas figuras em lugar dos pontos de inserção. Quando esses retângulos forem movimentados, a linha os acompanhará.



linha.P1=figureModel1.getPC();

Figura 5.9: Conexão entre figuras.

O framework JHotSea disponibiliza um conjunto de figuras primitivas que podem auxiliar na composição de novas figuras. As primitivas disponíveis correspondem aos modelos adotados pela própria linguagem Java, são elas: o retângulo, o círculo, a linha e o texto.

5.4 Comportamento do JHotSea

A grande dificuldade em se desenhar um editor gráfico reside na ausência de documentação sobre o aspecto comportamental dos editores. Outro agravante é a complexidade que envolve o uso de recursos gráficos nas linguagens orientadas a objeto pois quase sempre não são extensamente documentados.

O JHotSea é um framework de permite a criação tanto de aplicações como de Applets dependendo de como a superclasse *editor* seja invocada pelo editor do usuário. A classe *editor* é um applet, entretanto o método *open()* permite comportar-se também como uma aplicação comum. Os detalhes de sua utilização serão apresentados no próximo capítulo.

O JHotSea disponibiliza a classe *composite* (*model, view* e *constraint*) para para compor novas figuras.

Constraint

Cada figura possui uma série de propriedades que devem ser definidas em seu *figureConstraint*:

eraseable identifica se a figura, uma vez criada, pode ser apagada;

moveable identifica se a figura, depois de criada pode ser movida diretamente.;

selectable identifica se a figura é selecionável;

editable identifica a capacidade da figura de ser editada;

uniD identifica se a figura é criada a partir de 2 pontos, como por exemplo uma linha;

linkable identifica a capacidade da figura de apenas poder ser criada para ligar duas outras figuras;

connectable identifica a capacidade da figura de ser um ponto de ancoragem de uma figura do tipo *linkable*.

Operações

Para facilitar as operações com figuras, foram determinados os 6 atributos da classe tipo *abstractFigureConstraint* que, confrontados com os atributos da classe *drawingView* acessam os métodos da classe *figureModel* correspondente. A tabela 5.1 apresenta as regras para chamada aos métodos.

Método	**Atributos Constraints**						**Atributos View**	
	Editable	**moveable**	**UniD**	**linkable**	**Erasable**	**Connectable**	**Open Link**	**Selected**
EditModel	true							true
CreateModel				false			false	false
CreateModel			True	true		true	false	true
MovePC		true					false	true
FixP1			True	true		true	true	true
FixP2			False				true	false
EraseModel					true			true

Tabela 5.1: Métodos da classe *drawingView* para acesso aos modelos.

Observe que cada atributo pode possuir três estados: *true, false* e *null*. Os quadro em branco representam o estado *null*, representando os casos em que a verificação do atributo é desnecessária.

Essa tabela está implementada na interface *callFigure* que pode ser modificada convenientemente.

Para outra chamada diferente dessas, talvez seja interessante usar a classe *drawingView* como um *template* e acrescentar novos métodos.

Criação, edição e exclusão de figuras

Numa aplicação criada a partir do JHotSea, sempre que se clica em um ponto vazio da área gráfica, um objeto tipo *figureView* é instanciado e repassado à classe *drawingView*.

A classe *drawingView* inclui o objeto *figureView* ao vetor de figuras e aciona o *drawingModel* para que armazene o *figureModel* correspondente em um vetor.

A classe *drawingModel* instância um objeto *figureConstraint* associado.

A figura 5.10 apresenta um diagrama de sequência da criação de uma figura.

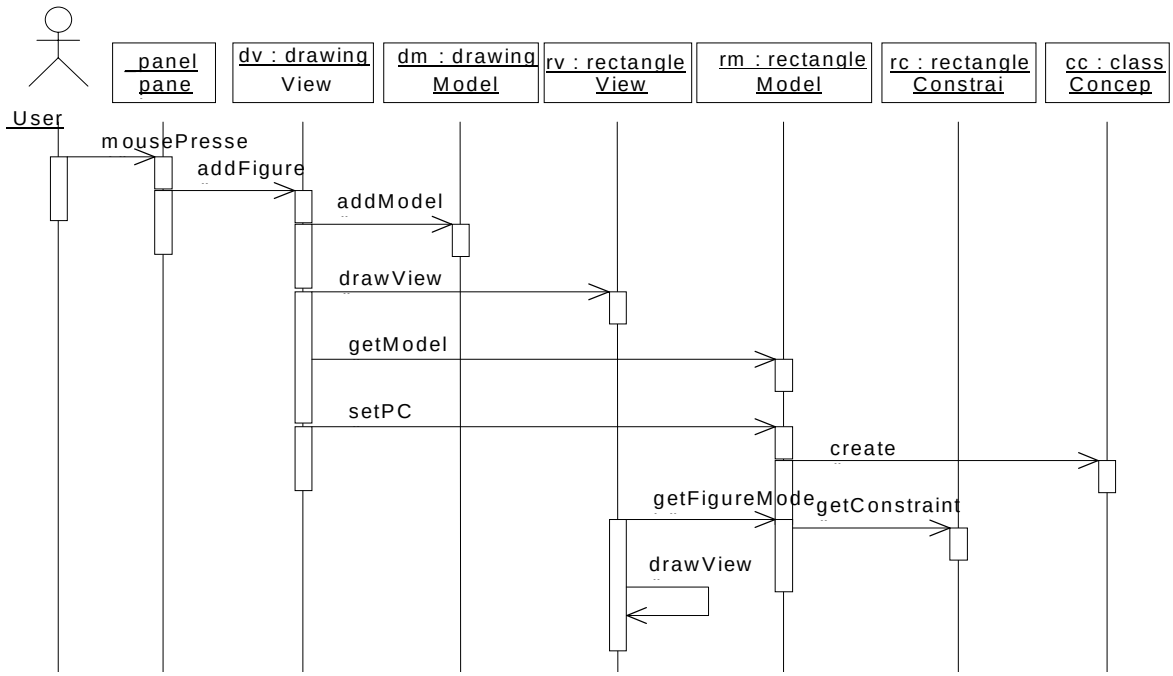


Figura 5.10 Diagrama de Sequência para criação de figuras a partir da classe *panel*.

A figura 5.11 apresenta o Diagrama de Sequência da edição de uma figura.

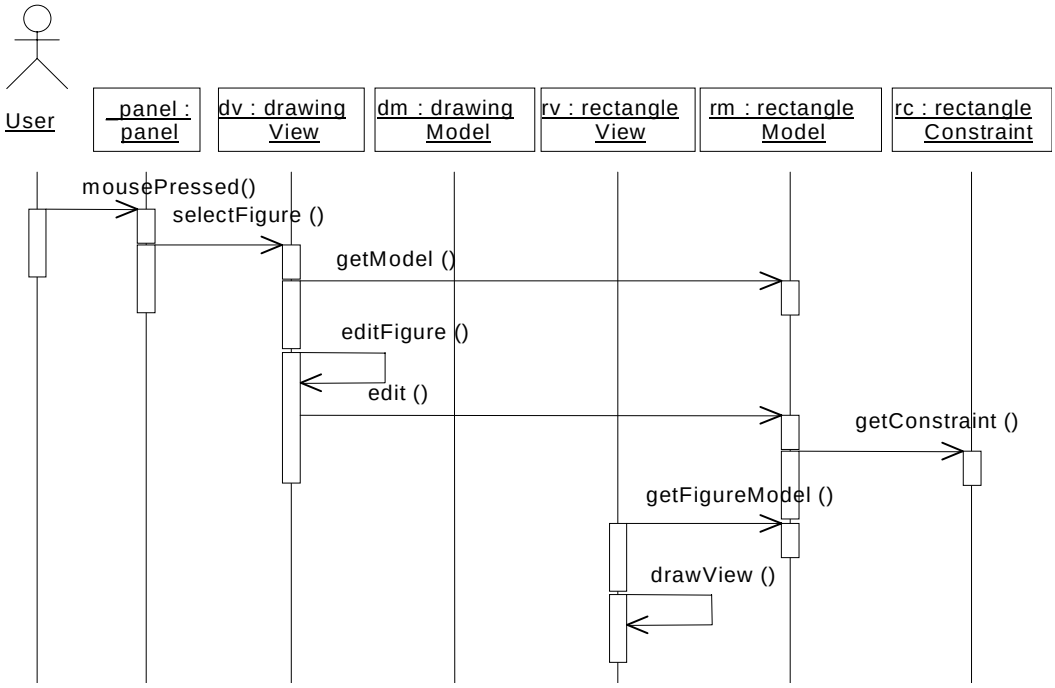


Figura 5.11: Diagrama de Sequência editar figura.

Observe que a edição ocorre quando uma figura é selecionada e a ferramenta correspondente também está selecionada.

A exclusão de uma figura realiza apenas a exclusão do objeto do vetor de figuras, o que pode ser visualizado no diagrama de sequência da figura 5.12.

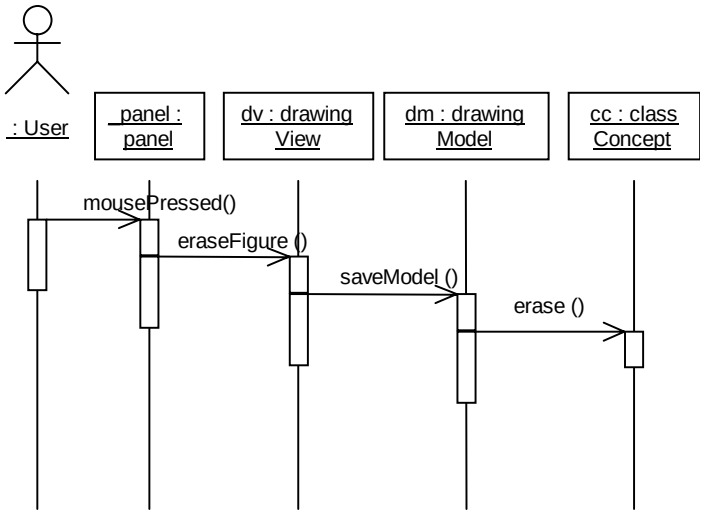


Figura 5.12: Diagrama de Sequência para excluir uma figura

Compartilhando em Rede

A estratégia adotada é tão somente de ambiente de distribuição e não de colaboração pois neste último caso, há que se considerar fatores como a velocidade de comunicação, no desempenho do aplicativo resultante. Assim um objeto pode ser compartilhado em apenas uma única direção. Portanto, é dispensável a discussão sobre tecnologias de suporte à colaboração como XML, JSP, Servlets, etc.,

O objeto a ser compartilhado entre aplicações clientes e servidores é o *drawingModel*. Como já foi apresentado mo item 5.1 deste capítulo, esta classe contém todos os objetos *figureModel* da aplicação. Então para que todas tenham a mesma apresentação basta que este objeto seja idêntico e que, se modificados por um cliente, todos os outros também sejam atualizados.

Por essa razão na classe *drawingView* quando uma figura é criada, uma referência de seu *figureModel* é passada ao *drawingModel* que o armazena num objeto da classe Vector[20].

A classe *client* do pacote *util* é a interface cliente para envio e recebimento de *sockets*. A classe *Server* tem comportamento híbrido atuando como cliente e servidor, conforme já discutido no capítulo anterior.

Uma abordagem mais simples também pode ser usada, usando somente a classe *client* para efetuar a carga de um objeto armazenado como arquivo no servidor Web para ser visualizado em um applet. O diagrama de sequência da figura 5.13 documenta essa operação.

[20] Vector – classe Java similar a um array, porém trata coleção de dados com recursos avançados.

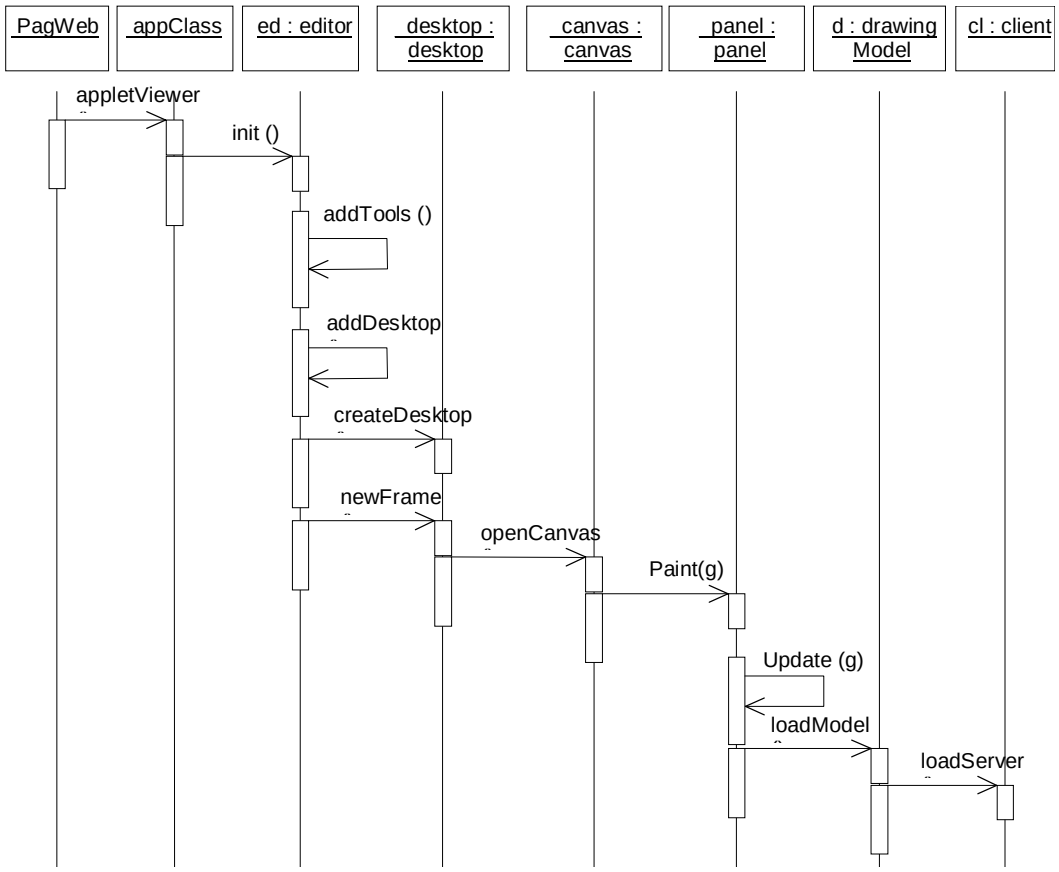


Figura 5.13: Diagrama de Sequência carga de figure a partir do servidor Web.

5.5 Implementação do Framework

O JHotSea foi implementado em Java na sua versão 1.3.1. O editor empregado foi FRED-*Framework Editor*[21], um editor específico para frameworks que utiliza o compilador Jike 1.3 da IBM. Esse compilador é mais rápido e portanto mais adequado para a fase de desenvolvimento [IBM,01]. Uma vez concluída a implementação optou-se por utilizar o compilador padrão da *Sun Microsystems*, existente no pacote JDK a fim de garantir a otimização de código, e de outras garantias do contrato do Java.

[21] FRED – Editor para framework desenvolvido pela Universidade de Helsinki-Finlândia disponível em http://practise.cs.tut.fi/fred/

Assim, a figura 5.14 apresenta as 6 principais classes do framework e suas superclasses Java correspondentes.

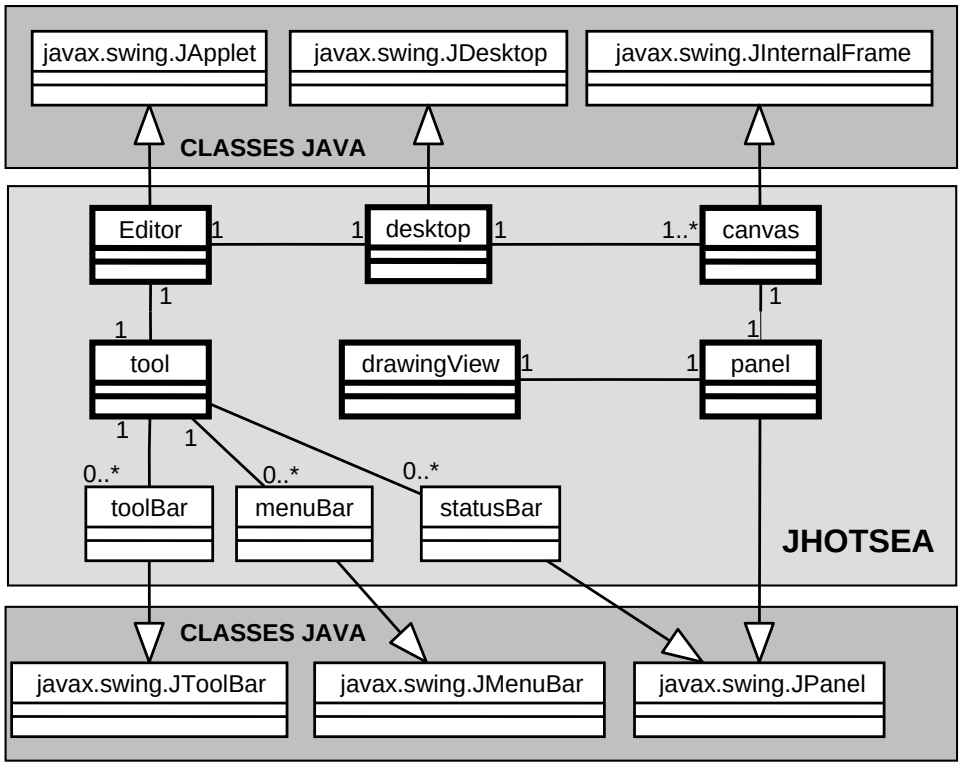


Figura 5.14: Classes principais do JHotSea e suas superclasses Java.

A classe *tool* não tem classe Java correspondente, apenas as classes de ferramentas a ela associadas (*toolBar*, *menuBar* e *statusBar*) é que possuem.

Observe que todas as superclasses utilizadas são do pacote Swing.

As classes principais do JHotSea, como se observa na figura 5.16, são subclasses de classes Java que possuem o mesmo conceito:

editor subclasse de *JApplet*, pode suportar *frames*;

desktop subclasse de *JDesktop* e pode instanciar muitos JInternalFrames;

canvas subclasse de *JinternalFrame*;

panel subclasse de *JPanel*;

Organização do Pacote

O JHotSea possui 27 classes distribuídas em 8 pacotes (*package* em Java). Entretanto, nada impede que o usuário crie pacotes adicionais caso seja necessário trabalhar mais conceitos além dos disponíveis (*view, model, constraint* e *concept*).

Os pacotes são eles:

main contém as classes principais do framework. São elas: *tool, editor, desktop, canvas, panel, drawingView e drawingModel, drawingConstraint;*

figureView contém as classes *View* do JHotSea. São elas: *abstractFigureView* e *compositeFigureView*. Aqui também as classes *View* da aplicação tais com *rectangleView, lineView*, etc.;

figureModel contém as classes *Model* do JHotSea. São elas: *abstractFigureModel* e *compositeFigureModel*. Aqui devem ficar também as classes *Model* da aplicação tais com *rectangleModel, lineModel,* etc.; a interface *callRepository* que contém as tabelas relacionando funcionalidades;

figureConstraint contém as classes *Constraint* do framework. São elas: *abstractFigureConstraint* e a *compositeFigureConstraint*;

document contém as classes que dão suporte a navegação (*path, memoryCell* e *environmentManager*) e as que gerenciam o registro de conceitos dos elementos visuais criados (*specification, elementSpecification, concept, conceptModel* e *typeObject*).

tool contém classes abstratas e concretas de ferramentas do framework. São elas: *abstractToolBar, abstractMenuBar* e *abstractStatusBar*;

util contém classes utilitárias para o framework. Abstraem elementos básicos como *color, file, font* e *iconRepository;*

webTool contém classes de serviço para web: a *client* e a *server*;

images é um repositório de figuras a serem utilizadas na barra de ferramentas;

samples pacote extra contendo algumas classes de aplicações para exemplificar o uso do framework;

A figura 5.15 apresenta o diagrama de pacotes do JHotSea.

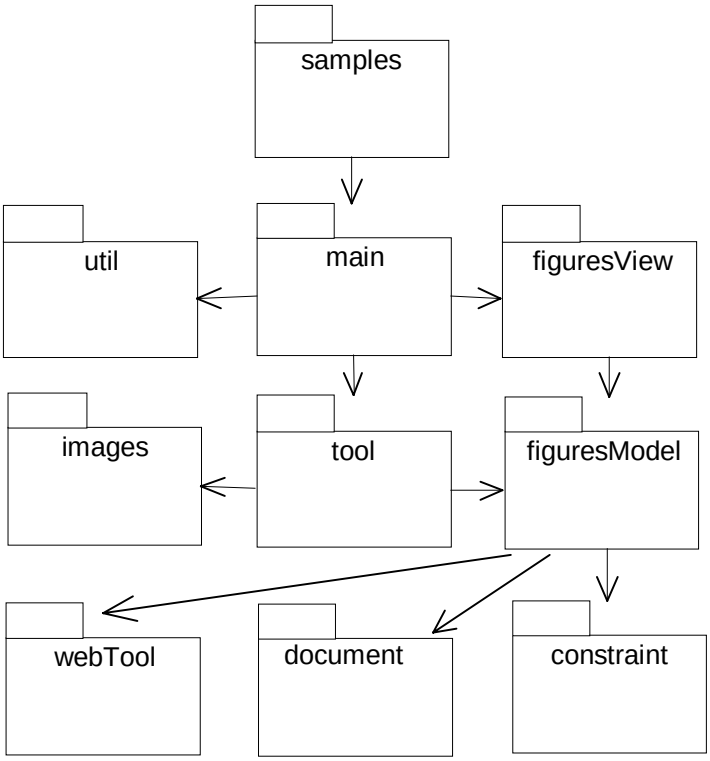


Figura 5.15: Diagrama de pacotes do JHotSea

Pela convenção proposta pela Sun [Deitel,00] para evitar colisão de pacotes com o mesmo nome em aplicações, estes foram precedidos hierarquicamente pela seqüência de pastas: br/ufsc/inf/sea.

Métodos de acesso

O acesso a atributos de uma classe é conhecido como métodos de acesso. São facilmente identificados pois são precedidos pelos prefixos *get* ou *set*.

Um procedimento usado para conferir flexibilidade ao framework é a introdução de métodos de acesso à objetos, instância de classes relevantes.

Assim, para criar novas funcionalidades em qualquer ferramenta basta acessar o modelo da figura invocando os métodos de acesso sucessivamente, partindo da classe *editor* (*getEditor()*) até se chegar ao modelo da figura (*getFigureModel()*).

O próximo capítulo trata sobre como implementar uma aplicação utilizando o JHotSea.

6. DESENVOLVENDO COM O JHOTSEA

O aprendizado do uso de um framework pode ser proporcionado de diversas maneiras. A disponibilidade de exemplos que contemplem o uso das principais funcionalidades do domínio contribui significativamente para o entendimento e uso do framework.

Apresenta-se a seguir roteiros para o desenvolvimento de duas aplicações utilizando o framework JHotSea. Os recursos deste framework não estão limitados aos apresentados nos exemplos, entretanto servem como ponto de partida para aplicações mais elaboradas.

6.1 Procedimentos para Criar uma Aplicação

A criação de uma aplicação comum exige que a classe do editor do usuário seja subclasse da classe *editor*, e dentro de seu método *main*() seja invocado o método *open*() como pode ser constatado no trecho de código 6.1 a seguir.

```
public class myEditor extends editor {
        static public void main(String arg[]){
          myEditor ed = new myEditor();
          ed.setTitle("SeaApp->");
          ed.setSize(500,500);     // Indica as dimensoes do frame da aplicacao
          ed.setStatusBar("Welcome to JHotDrawSea!");
          ed.open();
        }
}
```

(6.1)

Outros métodos específicos podem ser invocados dentro do método *main*, geralmente usados para configuração com editor como é o caso do *setSize*() e do *setTitle*() que, respectivamente configura o tamanho do *frame* a ser usado pela aplicação e indica o texto para a barra de titulo da aplicação.

A criação de um applet pode ser feita apenas criando a subclasse a partir da classe *editor*, pois dispensa a configuração da barra de títulos, como no trecho de código 6.2.

```
public class myAppletEditor extends editor {
    ...
}
```
(6.2)

Uma vez criada a classe applet, esta deve ser referenciada corretamente nas *tags* HTML da página Web conforme sugerido no Anexo I.

Criar figuras Novas

O JHotDraw possui um conjunto de figuras primitivas. Figuras diferentes ou uma combinação das já existentes podem ser criadas.

Para criar novas primitivas é necessário criar subclasses das classes abstratas *abstractFigureView*, *abstractFigureModel* e *abstractFigureConstraint* e complementar sua implementação dos métodos abstratos e do construtor. Uma boa prática consiste em examinar as classes semelhantes para entender a implementação. Um ponto importante na implementação do *constraint* é a definição das restrições de comportamento da figura em função das propriedades descritas na seção 5.4 deste trabalho (*editable*, *connectable*, etc).

Para criar figuras a partir das primitivas já existentes utiliza-se o padrão *composite* criando-se subclasse para as classes *compositeFigureView*, *compositeFigureModel* e *compositeFigureConstraint*. Na subclasse de *compositeFigureView*, instancia-se objetos das classes *figureView* correspondentes as primitivas que devem compor a figura.

Caso haja necessidade de modificar alguma característica do modelo instanciado, como por exemplo o ponto de inserção, pode ser feita sobrepondo os métodos das subclasses *compositeFigureModel* e *compositeFigureConstraint* correspondentes, que fazem a chamada ao método da instância da primitiva.

Como é possível observar, houve necessidade de criar uma primitiva, o *triangleView*, e também uma subclasse *heritView*. As classes *model* e *constraint* correspondentes não foram aqui apresentadas, mas devem ser criadas paralelamente.

Para figuras que utilizam o suporte a navegação basta sobrepor o método *navigation*() indicando o *model* do documento a ser vinculado. Assim, por exemplo, é possível abrir um diagrama de classes vinculado a uma figura *package* invocando o método *navigation*() a partir do método *edit*() da figura.

Agregar figuras a barra de ferramentas

O JHotDraw possui um conjunto de figuras primitivas. Figuras diferentes ou combinação das já existentes podem ser acrescentadas e disponibilizadas na barra de ferramentas a partir do seguinte roteiro**:**

a) criar uma arquivo de desenho tipo “gif” e o colocar na pasta “images”;

b) referenciar o arquivo na classe *iconRepository* do pacote *util*:

```
:
ImageIcon HERIT_ICON = new ImageIcon(IMAGES_DIR + "herit.gif");
:
```
(6.3)

c) criar um *figureView*, *figureModel* e se for o caso um *figureConstraint* para a figura (*heritView*,*lineModel* e *heritConstraint*, por exemplo);

d) instanciar na classe figureModel correspondente a figura, um objeto de uma subclasse de *concept* (classConcept, por exemplo) e sobrepor o método *setConcept*();

e) Para criar uma barra de ferramentas nova basta criar uma subclasse da classe *abstractToolBar* ou, caso desejar, pode usar a classe padrão *toolBar1*.

f) inserir o *button* no vetor de botões buttonToolBar da classe *toolBar1* do pacote *tool*, indicando o ícone a ser criado.

```
private JButton[] buttonToolBar={
        ...
         new JButton(iconRepository.HERIT_ICON),
}
```
(6.4)

g) referencie o *figureView* correspondente na estrutura case do método *getNewFigure* da classe *toolBar1*.

```
...
public abstractFigureView getNewFigure(){
    switch (iButton) {
        ...
        case 7: return new heritView();    //Create heritage figure
        ...                                                    (6.5)
```

A barra de ferramentas padrão (classe *toolBar1*) já disponibiliza as funções de seleção e de exclusão de figuras vinculadas aos eventos do mouse, gerenciados pela classe *panel*.

Agregar funcionalidades

O framework já apresenta as funcionalidades básicas de criar, mover, apagar e editar. Novas funcionalidades podem ser criadas a partir das chamadas de edição, modificando-se o método *edit* da classe *drawingView*. Para a inserção de operações que envolvam eventos do mouse ou combinação de teclas basta usar os métodos correspondentes disponíveis na classe *panel*.

Agregar opções de Menu

Para incluir uma opção na barra de menu estende-se a classe *abstractMenuBar* e referencia-se na classe criada a opção desejada e a operação correspondente. É possível também utilizar a subclasse já existente menuBar1. A forma de implementar segue a sintaxe Java especifica para o caso. As operações devem estar implementadas em métodos de classes do tipo Action[22] externas ou internas a qualquer classe.

WebTool

As classes *Client* e *Server* devem ser instanciadas na classe *drawingView* a fim de permitir o compartilhamento dos recursos em rede. Caso se queira apenas a leitura a

[22] Classe *Action* – Interface Java cujas subclasses são do tipo *ActionEvent* e dão suporte para que seus métodos sejam invocados mediante a ocorrência de eventos do hardware.

partir de servidor Web, sobrepor ou invocar o método *load*() da classe *Client* no método *updateModel*() da classe *drawingModel*. Em caso de compartilhamento, é necessário invocar o método *upload*() da classe *Server* no método *updateModel*() da classe *drawingModel*.

6.2 Exemplo 1: Editor para diagramas de classe

O Editor de Diagrama de Classes trata-se de um editor de gráficos estruturado simples e adotou-se um conjunto de ferramentas reduzido apenas para fim de comprovação da adequabilidade do framework proposto. Esta aplicação foi implementada *stand-alone*, utilizando o ambiente do próprio sistema operacional.

Ferramentas

Foram criadas três classes de ferramentas instanciadas na classe *tool*:

toolBar1 estendida a partir de *abstractToolBar* contendo botões referentes às 5 figuras e botões de seleção e exclusão;

menuBar1 estendida a partir de *abstractMenuBar* contendo opções do padrão Windows para arquivos, tais com *new*, *open*, *save*, etc.;

statusBar1 estende a partir da classe *abstractStatusBar*;

Figuras

As figuras *class* e *package* representam respectivamente uma classe e um pacote. São do tipo *connectable* e *editable.*

Foram criados três tipos de figuras dos tipos uniD e linkable: associação, herança e agregação. As últimas diferenciam-se quanto a figura associada. No primeiro caso um triângulo e no segundo um losango.

Todas as figuras foram implementadas com o uso do padrão *composite*, cada qual com seu *figureView* (classView, assocView, heritView e aggregView), *figureModel*, *figureConstraint* e *concept* correspondente.

O aspecto apresentado pela interface gráfica é mostrado na figura 6.1, aqui representando as classes associadas à classe *tool*.

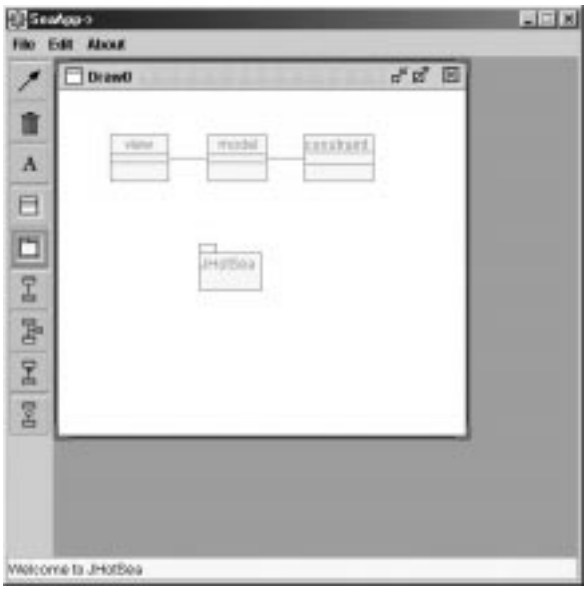


Figura 6.1: Interface Gráfica do Editor de Diagrama de Classe

6.3 Exemplo 2: Editor de diagramas de Seqüência

O Editor de Diagrama de Sequência também apresenta um conjunto restrito de ferramentas. Neste exemplo as figuras usam extensivamente o padrão *composite*. Este exemplo é implementado como *applet* e permite a carga de desenhos a partir de um servidor Web.

Ferramentas

Foram criadas três classes de ferramentas instanciadas na classe *tool*:

toolBar1 estendida a partir de *abstractToolBar* contendo botões referentes às 3 figuras primitivas e botões de seleção e exclusão; possui um conjunto de ferramentas para criar um editor de gráficos básico;

menuBar1 estende a classe *abstractMenuBar* contendo opções do padrão Windows para gerenciamento arquivos, tais com *new*, *open*, *save*, etc.;

statusBar1 estende a classe *abstractStatusBar*;

Figuras

A figura *object* representa um objeto de classe. É do tipo *connectable* e *editable.* Foi implementada sem *composite*. A classe *objectConstraint* limita o movimento do objeto apenas na horizontal dentro dos limites da área gráfica.

A figura *message* representa a mensagem de um objeto a outro. É do tipo *uniD* e *linkable* e foi implementada como *composite* entre as classes *link* e *text.* O Objeto *messageConstraint* organiza a inserção e exclusão entre mensagens já desenhadas.

A figura *autoMessage* representa uma mensagem passada ao próprio objeto. É do tipo *linkable.*

Cada uma das figuras possui seu *figureView* (*objectView, messageView e autoView*), *figureModel, figureConstraint* e *concept* correspondentes.

WebTool

Como foi implementado como *applet* torna-se necessária à escrita de uma página em código HTML par poder comportá-la. A implementação segue o roteiro proposto na seção 6.1

A figura 6.2 apresenta a interface gráfica obtida e mostra o diagrama de sequência da troca de mensagem entre dois objetos tipo *figureView* e *figureModel*.

Figura 6.2: Interface Gráfica do Editor de Diagramas de Sequência.

Observe que a aplicação está embutida em um navegador. O estabelecimento de parâmetros nas instruções HTML, especificamente nas *tags* tipo *code*[23], a fim de definirem a configuração do editor (dimensões, por exemplo) também é possível.

6.4 Considerações sobre as aplicações

O esforço de implementação envolvido pode ser avaliado com base nos códigos do Anexo I deste trabalho, referentes a criação das figuras: associação, classe, objeto e mensagem, das barras de ferramentas para gerenciar diagramas de classes (*toolbar1*) e diagrama de Sequência (*toolbar2*) e da barra de menus (*menubar1*) comum aos dois exemplos, constante no anexo desse trabalho.

Um número reduzido de *hot-spots* da ao framework a vantagem de permitir a criação de aplicações estendendo um numero reduzido de classes. Entretanto, pode se tornar um fator de redução da flexibilidade desse framework.

Como é possível observar, o desenvolvimento de uma aplicação sob o JHotSea trata de três pontos: figuras, ferramentas e a interface da aplicação. Apesar de poucos procedimentos, transferiu-se a flexibilidade para os objetos desses procedimentos. Assim o número, a complexidade e diversidade de figuras, por exemplo, pode ser definido dentro da estrutura de classes *figure*. É possível criar um editor básico apenas estendendo a classe editor, ou ainda mais complexos com diversas classes estendidas.

Esse aspecto passa a ser um diferencial em relação a outros frameworks pois tem pontos de flexibilidade bastante rígidos e definidos, como é o caso do HotDraw e JHotDraw.

As aplicações obtidas tiveram um desempenho satisfatório no que diz respeito à sua eficiência. Não apresentaram degradação alguma com o decorrer de seu uso, ocorrência comum em aplicações sob o HotDraw que se tornam mais lentas com o tempo de execução.

---

[23] *code – tag* HTML que permite a passagem de parâmetros do navegados a uma aplicação qualquer, inclusive ao próprio navegador.

7. CONCLUSÃO

A abordagem deste trabalho procurou explorar a viabilidade do uso de frameworks e componentes no desenvolvimento de aplicações que possuam embutidas ferramentas voltadas para a edição de gráficos estruturados. O framework JHotSea constitui um suporte flexível e extensível para a construção de ambientes de dessa natureza. Caracteriza-se pela capacidade de suportar a construção de ambientes que manipulem diferentes tipos de especificação de figuras e que disponham de diferentes conjuntos de funcionalidades.

A seguir, são apresentadas as principais contribuições obtidas neste trabalho, suas limitações e discute-se sobre futuras extensões ao framework desenvolvido, bem como futuros caminhos de pesquisa que podem ser seguidos utilizando os resultados do presente trabalho.

7.1 RESULTADOS OBTIDOS

Este trabalho produziu um conjunto de resultados que englobam os objetivos estabelecidos para o desenvolvimento de framework específico. Os principais são:

- O projeto do JHotSea partiu do reuso de abstração de um grupo especifico de frameworks conforme discorreu a seção 4.4. Entretanto, procurou alinhar conceitos e terminologias de edição gráfica com o de classes Java cujos conceitos fossem congruentes, conforme se pode verificar na seção inicial do Capítulo 6 deste trabalho. Espera-se com isso facilitar o entendimento do framework e consequentemente diminuir o tempo de aprendizado do seu uso.

- O framework produzido apresenta um *design* simples, contendo apenas 6 classes centrais (*frozen-spots*), número esse inferior apresentado pelos demais frameworks do grupo do HotDraw. Esta simplicidade também resulta em facilidade no aprendizado, otimizando o desenvolvimento de aplicações.
- A inclusão de suporte ao gerenciamento de subclasses *concept* conferiu mais flexibilidade ao modelo, pois a qualquer instante é possível verificar a estrutura do documento criado e seus elementos, desde que a implementação realize essa verificação. A análise do domínio produziu conceitos que puderam também ser encapsulados no framework
- O projeto reusa também as capacidades do applet Java, permitindo ser usado como componente cujas interfaces são tratadas por um browser. Assim, é possível criar uma página em HTML que possua a área gráfica em uma posição, referenciando o *drawingView,* e a barra de ferramentas em outra posição da página ou até mesmo em outro *frame* da pagina, e todos se referirem ao mesmo *model*.
- O uso da linguagem Java além de remover as restrições de plataforma[24] predispõe as aplicações obtidas a também executarem tanto sob sistema operacional como sob um browser com necessidade de poucas linhas de implementação para adequá-lo, e mesmo assim o corpo da aplicação está preservado.
- O uso da linguagem Java e particularmente da versão superior a 1.2 constitui-se também num diferencial em relação aos frameworks tratados no capítulo 4. Nenhum deles contempla esses recursos do pacote Swing.
- A adoção de métodos de acesso aos objetos instanciados a partir das classes relevantes amplia a disponibilidade de *hot-spots*.
- O JHotSea pode ser enquadrado como um caixa-cinza que explora as melhores vantagens de um desenho caixa-preta, na medida em que possui associação de classes de forma precisa e otimizada, e vantagens de um framework tipo caixa-

[24] plataforma – denominação que se dá ao conjunto de hardware e sistema operacional específicos.

branca disponibilizando mecanismos de herança para área gráfica, figuras e ferramentas.

7.2 Limitações

O JHotSea apresenta um conjunto de limitações em relação aos requisitos para um framework para o desenvolvimento aplicações que utilizem gráficos estruturados. A seguir são discutidos as limitações do framework JHotSea e os possíveis meios de superá-las.

A infra-estrutura de comunicação da Web impõe variações constantes na velocidade de transmissão de dados. Esse fator inviabiliza a edição gráfica compartilhada em tempo real. A necessidade de tráfego constante do *model* do desenho entre as maquinas virtuais[25] degrada ainda mais o processo face aos controles de integridade e colisão dos *sockets*.

Para servidores locais esse quadro é diferente, neste caso, uma configuração adequada de prioridades e contratos pode permitir a edição compartilhada entre vários usuários. A simples edição por uma máquina e a visualização por outros usuários é factível, já que a sua atualização em tempo real ou não, não causaria desconforto ao usuário.

O número de figuras primitivas disponíveis é relativamente pequeno. Entretanto, como podemos observar, o uso do padrão *composite* permite desenhar novas figuras a partir das figuras primitivas especificadas, conforme apresentado na seção 5.3 deste trabalho.

A necessidade de instalação do *plug-in* Java2, quando a plataforma for Windows, é também um fator que pode limitar o usuário, pois demanda de certos procedimentos de instalação e de configuração, podendo comprometer sua eficiência. Em plataformas mais antigas sua execução pode se tornar lenta.

---

[25] máquinas virtuais – ambiente operacional que encapsula o hardware real fazendo-o se comportar como outro com características diferentes (registradores, memória, etc.).

7.3 Trabalhos futuros

O framework JHotSea constitui um suporte para a construção e desenvolvimento de software específico e que deve ser estendido em futuros esforços de pesquisa.

Sua evolução permitiria ampliar o domínio da aplicação a fim de contemplar novas ferramentas, e poderão se constituir em futuras extensões do framework. A seguir são descritos trabalhos e linhas de pesquisa que podem ser suportados pelo trabalho ora desenvolvido.

O comportamento de aplicações sob esse framework embutidas em scripts para navegadores como o JSP é algo interessante pois dispensaria a instalação do *plug-in* Java 2 que, como já foi exposto, é um fator de limitação.

A avaliação de outras arquiteturas de suporte a distribuição tais como o CORBA e COM, podem agregar flexibilidade ao projeto além de permitir a integração com os ambientes que já possuam suporte para elas.

A inclusão de novos recursos que demandem uma reavaliação dos métodos de eventos disponíveis principalmente pela classe *panel* pode permitir recursos ainda mais elaborados como o manuseio de imagens e animação gráfica.

Por fim, a utilização de frameworks embutidos no próprio sistema operacional, como ocorre com o .Net aliado ao fato de ter sido escrito para plataforma neutra abre espaço para a especulação sobre a integração total ou parcial a algum ambiente operacional existente, o que poderia resultar em ganhos cuja grandeza ainda não é possível dimensionar.

7.4 Considerações Finais

A busca por estruturas de suporte ao desenvolvimento de ambientes gráficos que possuam otimização de recursos, facilidade no desenvolvimento, flexibilidade na adição de funcionalidades e integração com outros ambientes como a Web, é cada vez maior. Os frameworks se apresentam com recursos de reuso que podem atender a esses requisitos.

A flexibilidade em frameworks é obtida com intenso reuso (em todos os níveis) e por uma construção de classes de simples entendimento, mas abrangentes em relação a um domínio especifico.

Os recursos de implementação contribuem para a flexibilidade do componente disponibilizando recursos eficientes para representar a abstração através de códigos. Uma linguagem inadequada para implementar a solução é um fator restritivo. A linguagem Java permite a implementação adequada para o caso, face a diversidade de recursos.

O uso do ambiente Web e a adoção de construções de classes que permitam registrar o conceito de elementos gerados ampliam a flexibilidade do framework para o domínio em estudo.

Todos os frameworks analisados buscaram modelar abstrações com formalismos conceituais já existentes (grupos) a partir de variações apresentadas, como as disponibilizadas pelas linguagens de implementação. No entanto, a maioria desses modelos se restringe a descrever como e o que modelar no ambiente gráfico, utilizando-se quase sempre do formalismo de origem.

Observa-se que alguns modelos já atendem, em maior ou menor grau, aos requisitos de modelagem identificados. Acredita-se que a importância do JHotSea está no entendimento de que um framework para suporte a aplicações gráficas transcende ao simples domínio de interfaces gráficas, mas deve contemplar o desenvolvimento de aplicações que atuem como interface ambiental. Para o caso, a portabilidade entre os ambientes gráficos, Web, operacional e ao conceitual (ontologia) representa uma conquista importante, enquanto os demais aspectos são consequências dessa visão.

ANEXO I

RECURSOS DA LINGUAGEM JAVA

Java é uma linguagem de programação que suporta o paradigma de orientação a objeto, razão pela qual se adequa ao desenvolvimento de frameworks.

A forma de implementar os componentes e framework em Java apresenta vantagens em relação a outras linguagens pela forte orientação a abstrações conferida pela sua equipe de arquitetos da linguagem. Isso pode ser evidenciado com relação às interfaces gráficas onde muitos objetos do mundo real têm uma classe correspondente em Java.

A abordagem a ser apresentada neste capítulo procura documentar os recursos do Java que contribuíram de forma relevante no projeto do framework gráfico. Este capítulo se encerra discutindo a forma de implementação das abstrações em seu nível mais elevado: padrões e frameworks.

A Linguagem Java

Proposta em 1991 por James Gosling, inicialmente com um sistema operacional para uso em aparelhos eletrodomésticos a Oak, como era conhecida, ganhou notoriedade pela sua escolha como padrão para aplicações em ambiente Web, sendo incorporada nos principais *browser*s a partir de 1995 [Wutka,96].

Características da linguagem

É uma linguagem orientada a objetos desenvolvida pela *Sun Microsystems*. O Java reaproveita características de linguagens como Objetive-C, Modula-3, Smaltalk e principalmente do C++, suprimindo-lhe seus recursos de difícil implementação, tais como ponteiros e pré-processamento.

A linguagem Java foi projetada para ser pequena, simples e de arquitetura neutra, portável para diversas plataformas, tanto o código fonte como os seus binários. Seu compilador gera um código independente da plataforma, o *byte-code* que, *e*m tempo de execução é interpretado por uma máquina virtual local. Para que uma aplicação seja executada num ambiente é necessário apenas que possua uma máquina virtual específica para aquela plataforma [Gosling,95].

Java suporta programação concorrente e permite a utilização de programas em processamento paralelo real com a criação de diversos *threads*[26] de execução, desta forma uma plataforma paralela real poderá executa-lo em seus processadores sem a necessidade de fragmentar o código[Wutka,96].

Conserva quase toda a sintaxe do C++ em estruturas, tipos, operadores e outros. Possui tratamento de exceções de alto nível e realiza *garbage collection*[27] para otimizar o uso da memória. Adota padrão Unicode-16bits para o tipo *character* [Sun,01].

---

[26] *thread* - unidade de processamento, caracterizada por uma tarefa completa e independente, característica imprescindível para processamento paralelo real.

[27] *gabarge colletion* – suporte da linguagem para otimização de memória, liberando os endereço de variáveis, em tempo de execução, quando não for mais utilizada.

Uma máquina virtual Java possui suas próprias pilhas e registradores para uso de seus programas. Ela por sua vez é quem manipula diretamente os registradores e pilhas reais, aspecto esse que lhe confere extensa segurança contra *hacking* intencional ou por erro [Sun,01].

As classes do Java estão organizadas numa coleção de pacotes que permitem a implementação de aplicações mais comuns alem de disponibilizar componentes para aplicações especificas [Sun,01].

Applets X Aplicações

Uma aplicação Java pode ser desenvolvida para ser executada de duas formas: como aplicação pura ou como uma aplicação *browser*-integrada.

No primeiro caso a aplicação é executada diretamente na máquina virtual local, distribuída a partir de um servidor qualquer. No segundo caso, é necessário que a máquina local possua um *browser* instalado para servir o ambiente de máquina virtual. O applet, código da aplicação, é executado a partir de uma página Web após uma requisição de *download* para um servidor [Wutka,96].

É possível criar aplicações híbridas, applets que executam também como aplicações *stand-alone*[28]. Para tal basta criar um método que instancie a própria classe (applet) e execute seus respectivos métodos *start()* e *init()*. Não é possível executar o inverso, uma aplicação como se fosse um applet, pois este possui mecanismos de segurança que uma aplicação comum não tem implementado [Deitel,00].

Recursos Gráficos

Os recursos gráficos disponíveis pela linguagem permitem não só a criação de GUI's como também a representação de figuras primitivas (*shapes*). Esses recursos

[28] *stand-alone* – neste caso, aplicações isoladas, não distribuídas, instaladas e executadas no mesmo hardware

estão disponibilizados pelos pacotes *java.awt* e *javax.swing* Este último é uma extensão do primeiro e permite especificar elementos gráficos mais elaborados.

Componentes gráficos

A classe *Component* é a classe da qual descendem várias classes de objetos para a construção de interfaces gráficas do usuário, tais como: *Button*, *Label, TextArea*, *TextField*, *Canvas*, etc. Ela é uma das classes que contém o objeto *Graphics*, específica para desenhos em tela.

Arquitetura de um *container* Swing

Classes do pacote *java.swing* possuem *containers* em sua arquitetura, ou seja, permite que objetos de outras classes seja empilhados em sua estrutura através do método *add()* [Sun,01a].

O pacote Swing contém 4 tipos de *container* de alto nível: *JFrame*, *JDialog*, *JWindow* e *JApplet* ; possui também um *container* de baixo nível: o *JinternalFrame* . Essa diferença de nível está na quantidade de recursos disponíveis em função de suas superclasses [Armstrong,01].

Os componentes *java.swing* foram estendidos a partir dos componentes do pacote *java.awt*, com exceção do *JInternalFrame*, conforme apresenta a figura A.1.

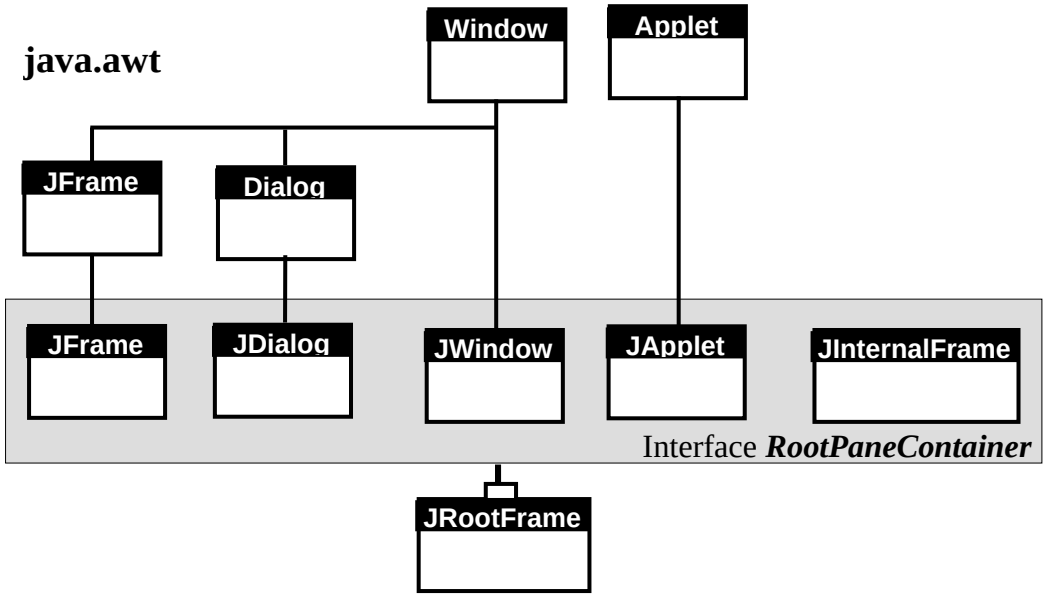


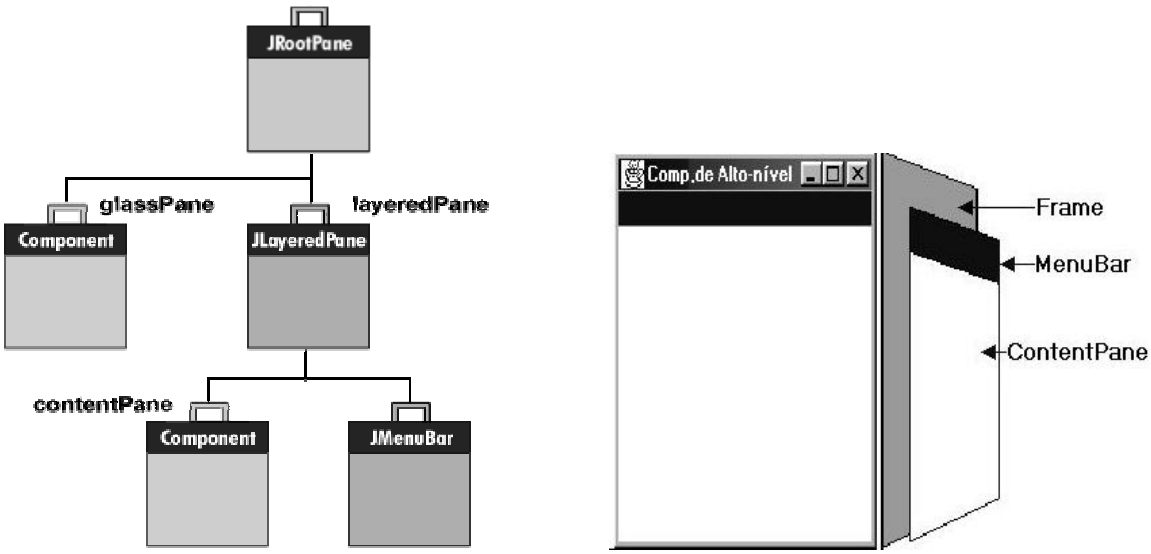
Figura A.1: *Containers* Swing [Sun,01a].

Todos os *containers* Swing implementam a interface *RootPaneContainer* que define a classe *JRootPane*. Esses cinco componentes delegam suas operações a *JRootPane*. Um objeto *JRootPane* é obtido através do método *getRootPane*(). A compreensão desse mecanismo permite empilhar painéis, quadros, janelas, etc.

Observa-se na figura A.2a que o objeto *JRootPane*, abstrai um painel raiz, com um painel transparente à frente (*glassPane*) e um *container* (*layeredPane*), permitindo empilhar outro *container* (*contentPane*) e um *menuBar*.

A estratégia de empilhar *container* sobre *conponent* constitui-se num *pattern* proprietário do Java. Tem vantagens sobre a simples associação por estruturar a associação entre classes dispensando o uso sistemático de vetores, o que tornaria seu controle de difícil implementação.

Relacionando uma classe *JFrame* da figura A.1 com o seu *JRootPane* da figura A.2a e seus elementos visuais da figura A.2b é possível entender como implementar um protótipo de GUI através das mais diversas classes Swing.



(a) Arquitetura [Sun,01a] (b)Componentes visíveis

Figura A.2: Arquitetura para uso de componente de alto nível

Uma conclusão importante sobre essa arquitetura é que a própria linguagem já incorporou muita das abstrações inerente a GUI de uma aplicação convencional, como é o caso da barra de menu. Este aspecto é relevante para o desenho de novos frameworks.

Ambiente gráfico

A área gráfica pode ser estabelecida através de um objeto *Graphics* e referencia a área gráfica da aplicação quando declarada na assinatura do método *paint()*.

A API Java2D fornece recursos gráficos bidimensionais avançados para gráficos. Esta API inclui recursos para processamento *line art* para gráficos.

A classe abstrata *Graphics2D* é uma subclasse de classe *Graphics* e só pode ser criada por coerção (*downcast*), como no trecho de código A.1, a seguir [Deitel,00]:

```
:
Graphics2D g2 =(graphics2D) g;
:                                                    (A.1)
```

Double buffering

Quando uma aplicação executa um desenho a lentidão da montagem deste sobre a área gráfica pode apresentar um efeito desagradável onde parte da área gráfica está com um desenho anterior e a outra sendo "construída" pelo novo desenho.

Para evitar esse efeito usa-se a técnica do *double buffering* onde o novo desenho é desenhado em um objeto *off-screen*[29] e depois de concluído substitui completa e instantaneamente a área gráfica antiga [Deitel,00].

O trecho de código A.2 exemplifica essa técnica:

```
private BufferedImage offScreenCanvas = null;   // Cria área off-scrreen
  :
public void paint(Graphics g) {
      // Cria objeto Graphics2D por coerção a partir de um objeto Graphics
      // objeto g2 possui a referencia da GUI atual (on-screen)
      Graphics2D g2 = (Graphics2D)g;
      // invocar aqui metodos que desenhe sobre offSreenCanvas
      g2.drawImage(offScreenCanvas, 0, 0, this); // Torna on-screen
}                                                    (A.2)
```

[29] off-screen – não é construído diretamente sobre a GUI, mas em uma variável ou objeto que o represente.

Delegação de Eventos

Existe uma diferença bastante grande entre programas baseados em janelas e programas baseados na console. Em um programa com saída para a console, o usuário dispara a execução e o próprio código determina a sequência dos eventos, onde geralmente tudo é pré-determinado. Já em uma aplicação baseada em janelas (ou um applet), a operação do programa é conduzida pela GUI.

A seleção de botões ou ícones usando o mouse ou o teclado causa ações particulares. Supondo-se que um usuário clique um botão em uma interface gráfica, o botão em questão será considerando a fonte deste evento, enquanto que o evento gerado a partir do clique do mouse será associado com o objeto *JButton* (que representa o botão na tela) nos programas escritos em Java[Eckel,00].

Um evento possui sempre um objeto fonte, neste caso o objeto *JButton*. Quando o botão é clicado, será criado um novo objeto que representa e identifica este evento, neste caso um objeto do tipo *ActionEvent.*

Este objeto conterá informações a cerca do evento e sua fonte. A figura A.3 ilustra esse modelo.

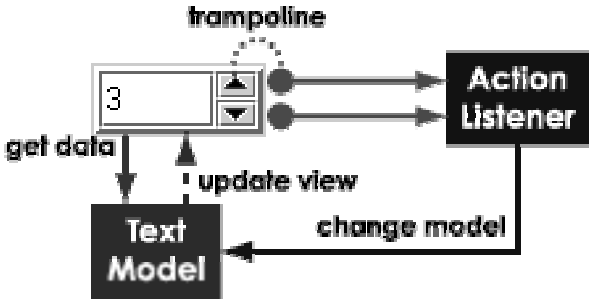


Figura A.3: Modelo de delegação de Eventos[Eckel,00]

Qualquer evento passado para um programa Java será representado por um objeto evento particular, sendo passado como argumento para o método que tratará o evento. O objeto evento correspondente ao clique no botão, será passado para um objeto *listener*, previamente registrado, que seja interessado neste tipo de evento. Um *listener* é também chamado de destino de um evento. Um objeto do tipo *listener* pode “escutar” eventos de um objeto particular (um botão, por exemplo) ou de um grupo de diferentes objetos , como os itens de um menu[Eckel,00].

O modelo de delegação de eventos, como é chamado esse processo, implementa o funcionalmente padrão *Observer* e consequentemente o MVC de uma maneira diferenciada pois a origem do elemento de controle está sempre no elemento vista. Qualquer mudança ou ação na interface gráfica pode gerar um evento.

Classes Internas e Anônimas

As classes internas são recursos da linguagem Java que permitem especificar ou instanciar uma classe dentro de outra classe ou de um método.

Esse recurso é importante quando se deseja reduzir o esforço de implementação, pois a classe interna pode chamar métodos da externa e vice-versa sem a necessidade de instanciação[Harold,00].

Observe no trecho de código A.3, o *metodoExterno* cria uma subclasse anônima da classe *classeExterna2* e o *metodoInterno2* poderia até ser uma sobreposição.

```
class classeExterna1{
        class classeInterna1 {
                public int campo;
                public void metodoInterno1(){}
        }
        public void metodoExterno( new classeExterna2(){
                public void metodoInterno2(){}
        }
}
```
(A.3)

Classe externa age como um pacote para várias classes internas. Entretanto o compilador cria classes separadas em tempo de compilação.

A implementação de classes internas gera ponteiros seguros apontando para métodos localizados em classes internas e confere flexibilidade para desenvolver objetos descartáveis.

Particularmente, no tratamento de eventos é a estratégia mais usada pois dispensa a criação de subclasse *action*, sua instanciação e chamada de um método específico.

Recursos Web

O uso da web como ambiente de computação colaborativa é uma das soluções para compartilhar objetos. A abrangência global deste ambiente o credencia a ser usado em aplicações que queiram se beneficiar deste alcance [Eckel,00]. A seguir discute-se os recursos do Java mais relevantes para o presente trabalho.

Applet

Uma applet possui vários recursos gráficos para a apresentação de dados na tela, sendo que os recursos mais comuns de apresentação de dados, as saídas padrões do Java, não funcionam no browser, pois são recursos não gráficos.

Uma applet possui várias implementações de segurança que um aplicativo Java não tem. Enquanto que em um aplicativo normal em Java é possível ler, gravar, alterar arquivos e configurações do sistema, a linguagem Java impede que uma applet faça livremente qualquer tipo de acesso a recursos físicos do sistema local.

O browser, antes de carregar uma applet, carrega uma classe chamada *appletviewer*, que é uma *thread*. A classe *appletviewer* recebe como parâmetro a applet a ser carregada e que em seguida invoca métodos referidos anteriormente. Estes métodos são chamados necessariamente na seguinte ordem: *init*() – na carga da applet, *start*() – todas vezes em que a applet se torna visível, *stop*() – todas as vezes em que a applet se torna invisível, *paint*() – quando a applet precisa ser redesenhada e pode ser chamado explicitamente através do método *repaint*()[Wutka,96].

O método *paint*() recebe como parâmetro uma instância da classe *Graphics*. A classe *Graphics* é utilizada para saídas gráficas no *browser*. Todos os métodos desta instância imprimem na área de visualização da applet. Tudo o que não for impressão gráfica não sairá na applet, mas ficará na área de saída padrão do sistema.

Applets em Páginas HTML

Uma applet é executada a partir de uma página precisa ser especificada dentro de uma *tag*[30] do HTML. A sintaxe empregada é bastante simples e especifica a localização da applet no servidor e também a região da interface gráfica disponível para sua apresentação, e pode ser expressa genericamente como no trcho de código A.4:

```
:
<HTML>
<APPLET CODE=teste.class WIDTH=300 HEIGHT=100 >
</APPLET>
</HTML>
:
```
(A.4)

Applets que usam a versão do Java 1.2 ou superior necessitam de tags mais complexas. O pacote JDK 1.2 ou superior disponibiliza uma aplicação chamada HTMLConverter que permite a adequação de códigos HTML para executar applets dessa versão.

A execução de applets a partir do Java 1.2 exige também que seja instalado um *plug-in* junto ao navegador. Sem esse procedimento, a página apresentará apenas uma tarja cinza na região onde seria apresentado o applet.

Compartilhamento de objetos em Rede

O compartilhamento de objetos em rede (*networking*) constitui-se num recurso eficaz para o desenvolvimento de tarefas. Esse compartilhamento pode ser realizado em duas abordagens: sockets e com RMI (*Remote Method Invocation*) [Harold,00].

Sockets

Esta abordagem está baseada em TCP/IP *sockets* e *ServerSockets* e utiliza se*quência* de objetos ao invés de simples *strings.*

---

[30] *tags* - são identificadores usados em linguagem HTML (*HiperText Markup Language*) para definir seções dentro do programa que gera a página.

O servidor TCP/IP mantém a classe principal com o conteúdo a ser compartilhado. Ao se conectar, cliente recebe uma cópia da lista. A restante das alterações da lista de elementos é enviada através do servidor central como na figura A.4.

As mensagens são encapsuladas como *object* e f*acilmente* transmitidas por sequências de objetos (*objectStreams*).

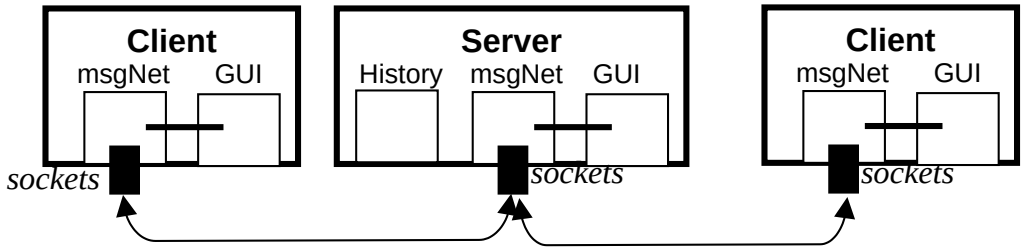


Figura A.4: Abordagem socket par compartilhar objetos.

Observa-se que a aplicação passa a comportar-se como um componente e utiliza os *sockets* como interface.

RMI

Nesta abordagem o servidor de *sockets* [Harold,00] é substituído por uma lista central (objeto remoto), as chamadas remotas são feitas sobre este objeto e os clientes se conectam ao sistema realizando chamadas remotas à lista central, como na figura A.5.

As manipulações subsequentes também são feitas como chamadas remotas a essa lista. A inviabilidade de serem feitas chamadas remotas da lista em resposta a cada cliente é um problema que pode ser contornado com um *pooling*.

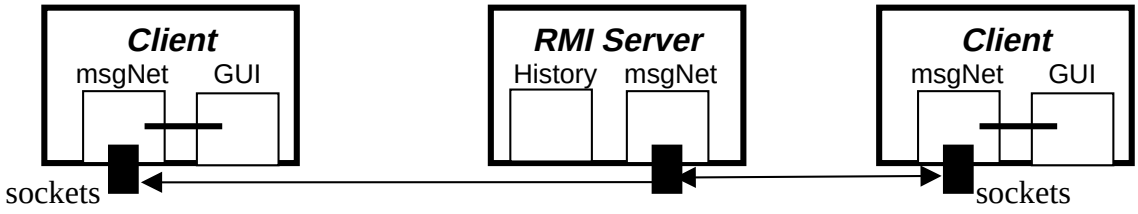


Figura A.5: Abordagem RMI para compartilhamento de objetos.

Comparando Sockets x RMI

A tabela A.1 identifica as vantagens e desvantagens entre as duas abordagens. Observa-se que a escolha do método depende da utilização que se vai dar ao modelo.

Se for para desenvolvimento corporativo o método *sockets* é mais adequado pois permite a atualização em tempo real. Se o objetivo é apenas *broadcasting* o RMI se torna a opção mais viável [Harold,00].

	Sockets	**RMI**
Prós	• Atualizações à lista são feitas em tempo real pelos clientes • Interface limpa	• clientes independentes da lista central • suporta conexão HTTP • pode operar através de *firewalls*
Contras	• clientes dependentes do servidor • dificuldade de implementar servidor *mutithreaded* • modelo não resistente a falhas da rede	• sistema de *polling* é lento • atualizações não são feitas em tempo real

Tabela A.1: Comparação entre as abordagens Sockets e RMI.

Serialização

O mecanismo de serialização de objetos é usado para transportar objetos entre máquinas virtuais. Implementar um objeto *serializable* permite a sua conversão numa sequência de bytes. A partir desta sequência é possível reconstruir uma cópia exata do objeto serializado em outra máquina virtual, conforme a fig. A.6. Essa operação também é conhecida como *marshaling*.

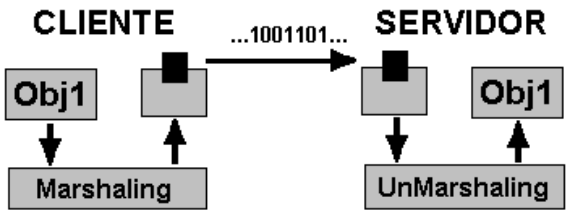


Figura A.6: Serialização de Objetos

Serializar objetos é gravá-los de maneira que no momento de restaurá-los seja possível recriar as instâncias e reconectá-las da maneira correta. Isto pode ser feito implementado-se a interface *serializable* do pacote *java.util.*

O próximo capítulo trata da concepção propriamente dita do framework para gráficos estruturados utilizando o Java como linguagem de implementação.

Implementando frameworks em Java

É possível implementar um framework em Java de diversas formas, a mais usual delas é cria-lo organizado em pacote e, ao criar uma aplicação invocá-lo através da cláusula *import* e estendendo as classes de interesse da aplicação.

```
import Framework1;
class classeAplicacao extends classe1 {
      ...
}
class class4 {
      class1 obj1=new classe1;
            ...
}
```
(A.5)

A figura A.7 apresenta um protótipo de framework contendo as classes *classe1* e sua subclasse *classe2*, encapsulada em um pacote framePac. A classe da aplicação gerada é a *classeAplicacao*. A notação usada aqui é a UML.

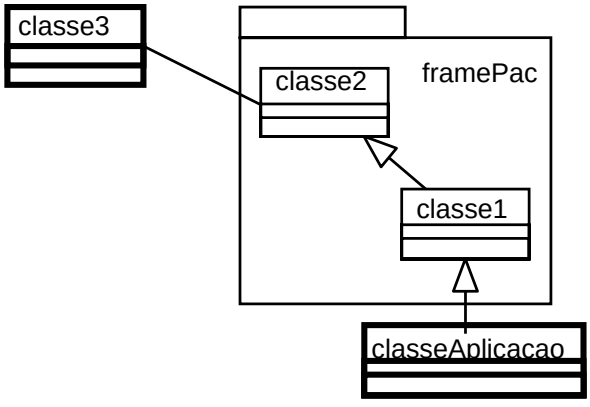


Figura A.7 Diagrama de classes de uma aplicação com framework

O código para esse diagrama de classes utiliza a sintaxe convencional do Java, sem a necessidade de nenhum recurso avançado. Para a criação do Framwork framePac teríamos como no trecho de código A.5 :

```
package framePac;
class classe2 {
        …
}
class classe1  extends class2 {
       …
}
```
(A.5)

Uma das formas de se disponibilizar um framework, JavaBean ou mesmo uma aplicação em Java é a utilização de arquivos JAR. Trata-se de um arquivo que disponibiliza todas as classes compactadas. A forma de compilar, reusar e executar componentes desse arquivo é definida pela Sun [Sun,00]. Em se tratando de applets, o uso de arquivos JAR é vantajoso por diminuir o tempo de carga pelo navegador.

Outras formas mais complexas podem ser usadas, como por exemplo, criar um objeto do tipo *package* a partir do framework e gerenciar seu uso na aplicação estendendo suas classes.

ANEXO 2

CONSTRAINTS

```
// Constraint da Figura Associação do Diagrama de Classes
package br.ufsc.inf.sea.figuresConstraint;
public class assocConstraint extends compositeConstraint{
        public assocConstraint(){
          addElementConstraint( new lineConstraint() );
          addElementConstraint(new arrowConstraint() );
          addElementConstraint( new textConstraint() );
          setConstraint (true,       // is connectable
                                false,// is editable
                                true,  // is erasable
                                false,// is linkable
                                true,  // is moveable
                                true,  // is selectable
                                true);// is unidimensional
        }
}

// Constraint da Figura Classe do Diagrama de Classes
// Nada foi descrito por que usa o Constraint padrao do CompositeConstraint
package br.ufsc.inf.sea.figuresConstraint;
public class classConstraint extends compositeConstraint{

}

// Constraint da Figura Objeto do Diagrama de Sequencia
package br.ufsc.inf.sea.figuresConstraint;
public class objectConstraint extends compositeConstraint{
        public objectConstraint(){
          addElementConstraint( new rectangleConstraint() );
          addElementConstraint( new LINEConstraint() );
          addElementConstraint( new textConstraint() );
        }

}

// Constraint da Figura Mensagem do Diagrama de Sequencia
package br.ufsc.inf.sea.figuresConstraint;
public class messageConstraint extends compositeConstraint{
        public packageConstraint(){
```

```
        addElementConstraint( new lineConstraint() );
        addElementConstraint( new arrowConstraint() );
        addElementConstraint( new textConstraint() );
      }
      public Point verifyPC(Point p){ return new Point(p.x, 30); } // Fixa o Y em 30
}
```

MODELS

```
// Model da Figura Associação do Diagrama de Classes
package br.ufsc.inf.sea.figuresModel;
import br.ufsc.inf.sea.figuresConstraint.*;
public class assocModel extends compositeModel{
        public assocModel(){
          setConcept ("assoc");
          addElementModel(new lineModel() );
          addElementModel(new arrowModel() );
          addElementModel(new textModel() );
          setFigureConstraint( new assocConstraint() );
        }
}

// Model da Figura Classe do Diagrama de Classes
package br.ufsc.inf.sea.figuresModel;
import br.ufsc.inf.sea.figuresConstraint.*;
public class classModel extends compositeModel{
        public classModel(){
          setConcept("class");
          addElementModel(new rectangleModel(0,0,60,30) );
          addElementModel(new rectangleModel(0,5,60,05) );
          addElementModel(new textModel(0,-10,10,10,"class") );
          setFigureConstraint( new classConstraint() );
        }
}

// Model da Figura Objeto do Diagrama de Sequencia
package br.ufsc.inf.sea.figuresModel;
import br.ufsc.inf.sea.figuresConstraint.*;
public class objectModel extends compositeModel{
        public objectModel(){
          setConcept ("object");
          addElementModel(new rectangleModel(0,0,60,30) );
          addElementModel(new lineModel(0,5,60,05) );
          addElementModel(new textModel(0,-10,0,0,"object") );
          setFigureConstraint( new objectConstraint() );
        }
}

// Model da Figura Message do Diagrama de Classes
package br.ufsc.inf.sea.figuresModel;
import br.ufsc.inf.sea.figuresConstraint.*;
public class messageModel extends compositeModel{
        public objectModel(){
          setConcept ("message");
          addElementModel(new lineModel(0,0,60,30) );
          addElementModel(new arrowModel(0,5,60,05) );
          addElementModel(new textModel(0,-10,0,0,"mess") );
          setFigureConstraint( new objectConstraint() );
        }
}
```

```
// View da Figura Associação do Diagrama de Classes
package br.ufsc.inf.sea.figuresView;
import br.ufsc.inf.sea.figuresModel.*;
public class assocView extends compositeView{
        public assocView(){
          addElementView(new lineView());
          addElementView(new arrowView());
          addElementView(new textView());
          setFigureModel( new assocModel() );
        }
}

// View da Figura Classe do Diagrama de Classes
package br.ufsc.inf.sea.figuresView;
import br.ufsc.inf.sea.figuresModel.*;
public class classView extends compositeView{
        public classView(){
          addElementView(new rectangleView());
          addElementView(new rectangleView());
          addElementView(new textView());
          setFigureModel( new classModel() );
        }
}

// View da Figura Objeto do Diagrama de Sequencia
package br.ufsc.inf.sea.figuresView;
import br.ufsc.inf.sea.figuresModel.*;
public class objectView extends compositeView{
        public objectView(){
          addElementView(new rectangleView());
          addElementView(new lineView());
          addElementView(new textView());
          setFigureModel( new classModel() );
        }
}

// View da Figura Mensagem do Diagrama de Sequencia
package br.ufsc.inf.sea.figuresView;
import br.ufsc.inf.sea.figuresModel.*;
public class messageView extends compositeView{
        public objectView(){
          addElementView(new lineView());
          addElementView(new arrowView());
          addElementView(new textView());
          setFigureModel( new arrowModel() );
        }

}
```

TOOLBAR

```
package br.ufsc.inf.sea.tool;
public class toolbar1 extends abstractToolBar implements ActionListener{
        private JButton[] buttonToolBar={
          new JButton(iconRepository.SELECT_ICON),
          new JButton(iconRepository.ERASE_ICON),
          new JButton(iconRepository.TEXT_ICON),
          new JButton(iconRepository.CLASS_ICON),
          new JButton(iconRepository.PACKAGE_ICON) ,
          new JButton(iconRepository.ASSOC_ICON),
          new JButton(iconRepository.TRIASSOC_ICON) ,
          new JButton(iconRepository.AGGREG_ICON) ,
          new JButton(iconRepository.HERIT_ICON) ,
        };

        private String[] textStatusBar={
          "Selecting a figure",
          "Erasing a figure...",
          "Drawing a text...",
          "Drawing a class figure...",
          "Drawing a pakage figure...",
          "Drawing a association reloship...",
          "Drawing a tri association reloship...",
          "Drawing a aggregation reloship...",
          "Drawing a heritage reloship..."
        };

        public abstractFigureView getNewFigure(){

          switch (iButton) {
                  case 0: return select;                  // Select
                  case 1: return erase;                   // Erase
                  case 2: return new textView();          //Create text
                  case 3: return new classView();         //Create Class
                  case 4: return new packageView();       //Create Package
                  case 5: return new assocView();         //Create Àssociation
                  case 6: return new triassocView();      //Create Triassociation
                  case 7: return new aggregView();        //Create Aggregation
                  case 8: return new heritView();         //Create Heritage

                  default:  return error;
          }
        }
}
```

```
public class toolbar2 extends abstractToolBar implements ActionListener{
        private JButton[] buttonToolBar={
          new JButton(iconRepository.SELECT_ICON),
          new JButton(iconRepository.ERASE_ICON),
          new JButton(iconRepository.TEXT_ICON),
          new JButton(iconRepository.OBJECT_ICON),
          new JButton(iconRepository.MESSAGE_ICON) ,
          new JButton(iconRepository.ATOMESS_ICON),
          new JButton(iconRepository.NOTE_ICON) ,
        };

        private String[] textStatusBar={
          "Selecting a figure",
          "Erasing a figure...",
          "Drawing a text...",
          "Drawing a object figure...",
          "Drawing a message figure...",
          "Drawing a automessage reloship...",
          "Drawing a note figure...",
        };

        public abstractFigureView getNewFigure(){

          switch (iButton) {
                  case 0: return select;                  // Select
                  case 1: return erase;                   // Erase
                  case 2: return new textView();          //Create text
                  case 3: return new objectView();        //Create Object
                  case 4: return new messageView();       //Create Message
                  case 5: return new automessView();      //Create AutoMessage
                  case 6: return new noteView();          //Create Note

                  default:  return error;
          }
        }
}
```

MENUBAR

```
package br.ufsc.inf.sea.tool;
public class menubar1 extends abstractMenuBar implements ActionListener{
	//define o menu
	private JMenu[] menu ={new JMenu("File"),new JMenu("Edit"),new JMenu("About")};

	//Definicao dos Itens de menu
	private JMenuItem[][] item= {
	     {new JMenuItem("New"), new JMenuItem("Open"),new JMenuItem("Save"),
	          new JMenuItem("Close"), new JMenuItem("Exit")},
	          {new JMenuItem("Color"), new JMenuItem("Font"), new JMenuItem("Border")},
	     };
	public void actionPerformed(ActionEvent e) {
	  String currentMenuItem=((JMenuItem)e.getSource()).getText();
	  if ( currentMenuItem == item[0][0].getText() ) {  // Menu Item New
	          _editor.getDesktop().newFrame();
	  }
	  if ( currentMenuItem == item[0][1].getText() ) {  // Menu Item Open
	          _editor.getDesktop().openFrame();
	  }
	  if ( currentMenuItem == item[0][2].getText() ) { // Menu Item Save
	          _editor.getDesktop().saveFrame();
	  }
	  if ( currentMenuItem == item[0][3].getText() ) { // Menu Item Close
	          _editor.getDesktop().getCurrentCanvas().closeCanvas();
	  }
	  if ( currentMenuItem == item[0][4].getText() ) { // Menu Item Exit
	          _editor.closeEditor();
	  }
	}
}
```

REFERÊNCIAS BIBLIOGRÁFICAS

[Alexander,77] ALEXANDER, Christopher, et. al., *A Pattern Language*, Oxford University Press, New York, 1977.

[Alexander,79] ALEXANDER, Christopher, *The Timeless Way of Building*, Oxford University Press, New York, 1979.

[Arango,91] ARANGO, G.; PRIETO-DIAZ, R.. *Domain analysis concepts and research direction*. In: PRIETO-DIAZ.;ARANGO, G. Domain analysis and software modeling. Los Alamitos: IEEE Computer Society Press, 1991.

[Assmann,97] ASSMANN, U., SCHMIDT, R. *Towards a model for composed extensible components*. In: FOUNDATIONS OF COMPONENT-BASED SYSTEMS WORKSHOP, 1997, Zurich. Proceedings. Zurich: [s.n.], 1997.

[Beck,87] BECK, Kent; Cunningham, Ward. *Using Pattern Languages for Object-Oriented Programs*, Technical Report nº CR-87-43, 1987, disponível na URL: http:// c2.com/doc/ oopsla87.html

[Bellur,98] BELLUR, U., *The Role of Components & Standards in Software Reuse*, Oracle. In: Workshop on Compositional Software Architectures, Paper012, Monterey, CA-USA, 1998.

[Booch,96] BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML -Unified Modeling Language Version 1.1* – Rational, September 1996.

[Booch,99]. BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML User Guide*. Reading: Addison-Wesley, 1999.

[Bosch,97] BOSCH, J. *Adapting object-oriented components*. In: SECOND INTERNATIONAL WORKSHOP ON COMPONENT-ORIENTED PROGRAMMING, (WCOP'97), 2., 1997,

Jyväskylä. Proceedings… Jyväskylä: [s.n.], 1997.

[Brant,95] BRANT, J. *Hotdraw* Urbana: university of Illinois at Urbana-Champaign, 1995. Master thesis.

[Brant,01] BRANT, J. *Hotdraw Home page.* University of Illinois at Urbana-Champaign, disponível em fevereiro de 2001 naURL www.cs.uiuc.edu/users/brant/HotDraw

[Buchmann,96] BUSCHMANN, F.; JÄKEL, C.; MEUNIER, R.; ROHNERT, H.; STAHL, M., *PatternOriented Software Architecture - A System of Pattern*s, John Wiley & Sons, 1996.

[Buhr,99] BUHR, R. *Understanding Macroscopic Behaviour Patterns in Object-Oriented Frameworks, with Use Case Maps,* Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1999.

[Burbeck,92] BURBECK, S. *Applications Programming in Smalltalk-80: How to use Model-View-Controller (MVC)* .disponível em janeiro de 1992 na URL http://st-www.cs.uiuc.edu/users/march/st-docs/mvc.html.

[Campo,97] CAMPO, M. *Visual Understanding of Frameworks through Instrospection of Examples*. Ph.D. Thesis. UFRGS, CPGCC, 1997.

[Cetus,02] *Cetus-link – OO Frameworks*. disponível em fevereiro de 2001 na URL: geolab.ccei.urcamp.tche.br/cetus/oo_frameworks.html

[Coad,92] COAD, P.; YOURDON, E.. *Análise Baseada em Objetos*. RJ, Campus, 1992.

[Deitel,00] DEITEL, H.M. *Java How to Program*, Prentice-Hall, 2000.

[DeRemer,76] DEREMER, F.;KRON, H.H. *Programming-in-the-large versus programming-in-the-small*. IEEE Transactions on Software Engineering. NY, v.SE-2.n.2, 1976.

[Eckel,00] ECKEL, Bruce *Thinking in Java*, 2nd edition, Revision 12, disponível em dezembro de 2000 na URL www.mindview.net

[Eckel,01] ECKEL, Bruce *Thinking in Patterns in Java,* edition 0.5a disponível em maio de 20001 na URL www.mindview.net

[Fayad,99] FAYAD, Mohamed; YASSIN,Amr. *Application Frameworks: A Survey*. *Domain-Specific Application Frameworks. Frameworks Experience by Industry*. Wiley & Sons, 1999.

[Fekete,01] FEKETE, Jean Daniel. *A Multi-Layer Graphic Model for Building Interactive Graphical Applications.* Université de Paris-Sud, *2001*.

[Froehlich,01] FROEHLICH, Garry; HOOVER, James;LIU, Ling;SORENSON, Paul.*Designing Object-Oriented Framework*. Department of Computing Science University of Alberta, Edmonton, Canadá, 2001

[Gamma,95] GAMA, E.; HELM, R.; JOHNSON, R.; Vlissides, J. *Design Patterns –Elements of Reusable Object-Oriented Softwar*e. Reading-MA, Addison-Wesley,1995.

[Goldberg,83] GOLDBERG, A. *Smalltalk 80 - The Language and it´s Implementation***,** Addiso-Wesley, Reading,Mass, 1983.

[Gomaa,94] GOMAA, D. *et al*. *Statecharts: a visual formalism for complex systems. Science of Computer Programming*, [s.n.],n.8,1987.

[Harold,00] HAROLD, Elliotte Rusty. *Java Network Programing*. CA: O'Reylli & Associates, 2000

[Gosling,95] GOSLING, J.; MCGILTON, H. *The Java Language Environment, Sun Microsystem*, CA,USA, October 95.

[Guarino,97] GUARINO, N. *Formal Ontologies and Information Systems*. In: FIRST INTERNATIONAL CONFERENCE (FOIS), 1., 1998, Trento, Itália. Anais... Trento: IOS Press, 1998.

[Guizardi,01] GUIZARDI G. *A methodological approach for reuse-oriented software development based on formal domain ontologies.* Master Dissertation, Computer Science Department, Federal University of Espirito Santo, 2000.

[Helton,98] HELTON, D. *The impact of large-scale component and framework application development on business*. In: INTERNATIONAL WORKSHOP ON COMPONENT-ORIENTED PROGRAMMING, (WCOP'98), 3., 1998, Brussels. Proceedings Brussels:[s.n.], 1998.

[Henninger,95] HENNINGER, S. *Developing domain knowledge through the reuse of project experiences*. In: SYMPOSIUM ON SOFTWARE REUSABIITY (SSR), 1995, Seatle. Proceedings.. Seatle:[sn], 1995.

[Hopson,96] HOPSON, K.C. *Developing professional Java applets*. Sams Net, 1996.Indianapolis, In, USA.

[IBM,01] IBM. *IBM Research Compiler Project*. Disponível em setembro de 2001 na URL : http://www.research.ibm.com/jikes/.

[Johnson,88] JOHNSON, Ralph E.; Foote B. *Designing Reusable Classes*. Journal of Object Oriented Programming – JOOP, 1(2):22-35, Junho/Julho 1988.

[Johnson,91] JOHNSON, Ralph E.; Russo, Vincent. *Reusing Object-Oriented Design*s. Relatório Técnico da Universidade de Illinois, UIUCDCS 91-1696, 1991.

[Johnson,92] JOHNSON, R. *Documenting frameworks using patterns*.SIGPLAN Notices, v.27, n.10,Oct.1992. trabalho apresentado na OOPSLA, 1992.

[Johnson,93] JOHNSON, R. *How to design frameworks*. 1993. Disponível por FTP anônimo em st.cs.uiuc.edu (dez.98).

[Johnson,97] JOHNSON, Ralph. *CS497 Lectures – Lecture 12, 13, 14 e 17*, disponível na URL: http://st-www.cs.uiuc.edu/users/johnson/cs497/notes98/online-course.html

[Johnson,02] JOHNSON, Ralph. *Comparing framewoks*. Mensagem pessoal enviada para o autor em 22 de fevereiro de 2002.

[Johnson,02a] JOHNSON, Ralph. *New in Frameworks*. Mensagem pessoal enviada para o autor em 12 de agosto de 2002.

[Leite,94] Leite, J.C.S.P. Engenharia de Requisitos. In: *Notas de Aula*, PUC-RJ, 1994

[Lorenz,94] LORENZ M, Kid, J. *Object-Oriented Software Metrics - A practical guide*. Englenwood Cliffs : Prentice Hall. 1984.

[Martin, 95] MARTIN; J. ODELL, J. análise *e Projeto Orientados a Objetos*. São Paulo, Makron, 1995.

[Masiero,01] MASIERO,P.; MALDONADO,J.;BRAGA, R.;GERMANO, Fernão; *Padrões e Frameworks de Software-Notas didáticas*, ICMC-USP, 2002.

[McIlroy,68] MCILROY, M.D. *Mass-produced software components*. In: NORTH ATLANTIC TREATY ORGANIZATION

CONFERENCE ON SOFTWARE ENGINEERING, 1968. Garmisch-Partenkirchen. Proceedings… Garmisch-Partenkirche: [s.n], 1968.

[Milne,02] MILNE, Wendy. *Modelling and Viewing*. disponível em outubro de 2002 na URL: http://www.dcs.ex.ac.uk/~wmilne/graphics/modelling.pdf

[Murer,96] MURER, T. *et al*. *Improving component interoperability*. In: Special Issues In Object-Oriented Programin, Workshop Of The Ecoop, 1996, Linz. Proceedings… Linz:[s.n.],1996.

[Neighbor,89] NEIGHBORS, J. *Draco: a method for engineering reusable software systems*. In: PRIETO-DIAZ, R.;ARANGO, G. Domain analysis and software modeling. Los Alamitos: IEEE Computer Society Press, 1991.

[Poli,01] POLI, Roberto. *Ontological Methodology*. disponível em agosto de 2001 na URL http://www.mitteleuropafoundation.it/RP/Papers/OntologicalMethodology.pdf

[Petri,62] PETRI, C. A., *Kommunikation mit Automaten. Bonn: Institut für Instrumentelle Mathematik*, Schriften des IIM Nr. 2, 1962.

[Pompermaier,97] POMPERMAIER L.; PRICE, R. T. *Um editor diagramático para desenvolvimento de sistemas de informação na internet*. SIMPOSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, (SBES), 12.,1998, Out.13-16: Maringá, Br-Pr, Anais. Maringá : Departamento de Informática/Universidade Estadual de Maringá, 1998.

[Pree,94] PREE, W. *Design patterns for object oriented software development*. Reading: Addison-Wesley, 1994.

[Pressman,82] PRESSMAN, R. *Sftware Engineering. A practioner´s approach*. [S.l.]: McGrawHill, 1982.

[Prieto-Diaz,87] PRIETO-DIAZ, R. *Domain analysis for reusability*. In: PRIETO-DIAZ, R.; ARANGO, G. Domain analysis for reusability. Los Alamitos: IEEE Computer Society Press, 1991.

[Riehle,00] RIEHLE, D. *Framework Design*, A Role Modeling Approach. PhD. Thesis. Swiss FIT- Zurich, 2000.

[Rolemodelsoft,01] *HotDraw for Java*, disponível em fevereiro de 2001 na URL http://www.rolemodelsoft.com/default.htm

[Silva,00] SILVA, R. P. *Suporte ao desenvolvimento e uso de frameworks e componentes*; Tese de Doutorado, UFRS/II/PPGC, Porto Alegre: março de 2000.

[Silva,00b] SILVA, R. P.; *Reuso* In: Transparências do curso de Engenharia de Software, Unirondon-Cuiabá/MT, junho de 2000.

[Silva,99]. SILVA, R. P.; PRICE L. T. *Suporte ao desenvolvimento e uso de Componentes flexíveis*, In: SIMPÓSIO BRASILEIRO D ENGENHARIA DE SOFTWARE, (SBES), 13., 199, Florianópolis. Anais. Florianópolis:[s.n.], 1999.

[Schmid,97] SCHMID, H. A. *Systematic Framework Design by generalization*. Communications of the ACM, V. 40, nº 10, p. 48-51, 1997.

[Sourceforge,02] *SourceForge Vector Graphics Foundry Frameworks*, disponível em janeiro de 2001 na URL vectorgraphics.sourceforge.net/ frameworks.php

[Sun,01] SUN Microsystems. *Java Tutorial*. disponível para download em http://java.sun.com/docs/books/tutorial/index.html. em janeiro de 2001.

[Sun,01a] SUN Microsystems. *The Swing Connection*. disponível para download em http://java.sun.com/docs/books/tutorial/ index.html. em janeiro de 2001.

[Szyperski,97] SZYPERSKI, C. *et al*. *Sumary of the seond International Workshop on Component-Oriented Programming*. In: INTERNATIONAL WORKSHOP ON COMPONENT-ORIENTED PROGRA-MMING, (WCOP97), 2.,1997, Jyväskylä. Proceedings … Jyväskylä.[s.n.], 1997.

[Taligent,94] TALIGENT. *Building object-oriented frameworks*. [S.l.]: Taligent, 1994, Write paper.

[Vlissídes,90] VLISSÍDES, J. *Generalized Graphical Editing*. Ph.D. thesis, Stanford University, June 1990.

[Weck,96] WECK, W. *Independently extensible component frameworks*. In: INTERNATIONAL WORKSHOP ON COMPONENT-ORIENTED PROGRAMMING, 1., (WCOP'96), 1.1996, Linz. Proceedings… Linz [s.n.]. 1996.

[Tai,01] TAI, Andy. *The GUI Toolkit, Framework Page*. Disponível em fevereiro de 2002, no endereço:www.atai.org/guitool/

[Wutka,96] WUTKA, W., *et al. Hacking Java*, QUE. 1996,