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

Editor Distribuído de Manipulação Colaborativa de Documentos Diagramáticos

por
Paulo Henrique Cayres

Prof. Dr. Roberto Tom Price
Orientador

Porto Alegre, junho de 2001.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Cayres, Paulo Henrique

Editor distribuído de manipulação colaborativa de documentos diagramáticos / por Paulo Henrique Cayres. - Porto Alegre: PPGC da UFRGS, 2001.

139 f. : il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2001. Orientador: Price, Roberto Tom.

1. Engenharia de *software*. 2. Sistemas distribuídos. 3. Editores colaborativos. 4. Colaboração. I Orientador: Price, Roberto Tom. II Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profa. Wrana Panizzi
Pró-Reitor de Ensino: Prof. José Carlos Ferraz Hennemann
Pró-Reitor Adjunto de Pós-Graduação: Prof. Philippe Olivier Alexandre Navaux
Diretor do Instituto de Informática: Prof. Philippe Oliver Alexandre Navaux
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

Aos meus pais Dalva e Florisvaldo e a minha esposa Cátia que sempre acreditaram na minha capacidade.

A meus irmãos Carlos, Luciana, Tatiana e Ana Lúcia por sempre estarem à disposição quando necessitava de ajuda.

Agradecimentos

Ao Prof. Roberto Tom Price, pelo apoio, compreensão, paciência e incentivo.

A profª Cláuda Maria Lima Werner pelas correções e sugestões de conteúdos que auxiliaram na melhoria e conclusão deste trabalho.

Aos professores da UFRGS que compartilharam seus conhecimentos e idéias durante este trabalho.

Aos funcionários da biblioteca da UFRGS que me acolheram dentro de suas dependências durante o desenvolvimento deste trabalho.

A todos os amigos da UNIDERP, em especial Emerson A. M. Corazza, que participaram deste trabalho, auxiliando e incentivando.

A UNIDERP pela oportunidade e apoio.

Demais familiares e amigos que colaboraram com o desenvolvimento deste trabalho.

Sumário

Lista de Abreviaturas **8**
Lista de Figuras **10**
Lista de Tabelas **12**
Resumo **13**
Abstract **14**
1 Introdução **15**
1.1 Proposta **18**
1.2 Organização do trabalho **19**
2 Editores e mecanismos de compartilhamento de documentos **21**
2.1 Editores Textuais e Diagramáticos **21**
2.1.1 Editores Dirigidos por Sintaxe 21
2.1.2 Editores Diagramáticos 23
2.2 Mecanismos de Compartilhamento **24**
2.2.1 Eletronic Data Interchange 24
2.3 XML (*Extensible Markup Language*) **26**
2.3.1 Principais características da linguagem XML 28
2.3.2 XML: Objetivos do desenvolvimento 29
2.4 Banco de dados cliente-servidor **30**
2.4.1 Funções dos computadores cliente e servidor 31
2.5 Frameworks **32**
2.5.1 Desenvolvimento de *Frameworks* 35
2.5.2 Frameworks no processo de construção de editores diagramáticos 37
2.5.3 Modelo de Interação MVC 37
2.5.4 Controlador 40
3 Padrões de projeto e de arquiteturas de aplicações *web* e mecanismos para construção de ambientes distribuídos e heterogêneos **42**
3.1 Padrões de projeto **42**
3.1.1 Padrão *Observer* 42
3.1.2 Padrão *Composite* 43
3.2 Padrões de arquitetura para aplicações *web* **44**
3.2.1 Thin Web Client 45
3.2.2 Thick Web Client 46
3.2.3 Web Delivery 48
3.3 Mecanismos para Construção de Ambientes Distribuídos e Heterogêneos **48**
3.3.1 Arquitetura de Aplicações em Camadas 48
3.3.2 CORBA – *Common Object Request Broker Architecture* 51
3.3.3 Servlets 53

3.3.4 RMI – *Remote Method Invocation*....56
3.3.5 Stream de I/O para arquivos....58

4 Uma proposta de arquitetura para Ambientes Distribuídos de Edição Colaborativa....62

4.1 Ambientes colaborativos....62
4.1.1 Principais atividades de um ambiente colaborativo....62
4.1.2 Co-autoria Colaborativa de documentos....67
4.1.3 Implementação....68
4.1.4 Flexibilidade....69
4.2 Sistemas Colaborativos....70
4.2.1 Necessidade de Comunicação....71
4.2.2 Mantendo Identificadores de Usuários....72
4.2.3 Informação de Estados Compartilhados....73
4.2.4 Desempenho....73
4.3 Modelo de Colaboração....73
4.3.1 Comentário....74
4.3.2 Inclusão....74
4.3.3 Exclusão....75
4.3.4 Substituição....75
4.3.5 Proposta....75
4.4 Metodologia do projeto....76
4.5 Requisitos da arquitetura do FreDoc....77
4.6 Arquitetura do FreDoc....78
4.6.1 Pacotes da arquitetura proposta para o desenvolvimento de Editores Distribuídos de Geração Colaborativa de Documentos....80
4.6.2 Descrição das principais classes implementadas na arquitetura do FreDoc....83
4.7 Camadas da Arquitetura do FreDoc....87
4.7.1 Considerações sobre a utilização de padrões projeto e arquiteturas de aplicações web na arquitetura proposta....88
4.8 Instanciando um Editor através do Framework FreDoc....89

5 Um estudo de caso: um editor distribuído de geração colaborativa de diagrama de classes UML....91

5.1 Características Gerais....91
5.2 Implementação do FreDocUML....94
5.3 Funcionalidade do Protótipo....95
5.3.1 Página para *Login* do usuário....98

6 Conclusão....107

6.1 Contribuições....108
6.2 Trabalhos Futuros....108
6.2.1 Suporte a diferentes idiomas....109
6.2.2 Geradores de código....109
6.2.3 Validade do modelo....109
6.2.4 Controle de Visão....109

6.2.5 Integração com outras ferramentas (importação/exportação de documentos)........110

Anexo A Implementação da Classe *FrameAccessScreen* 111

Anexo B Implementação da Classe *SimpleDrawingCanvas* 119

Anexo C Diagrama de caso de uso: Acesso ao editor via *Web* 129

Anexo D Diagrama de Seqüência: Acesso ao editor via *Web* 130

Anexo E Diagrama de Seqüência: Ativar ferramenta de edição 131

Anexo F Diagrama de Seqüência: Inserir classe no diagrama 132

Anexo G Diagrama de Seqüência: Inserir nota no diagrama.................133

Bibliografia 134

Lista de Abreviaturas

API	*Application Programming Interfaces*
CGI	*Common Gateway Interface*
CORBA	*Common Object Request Broker Arquiteture*
CPU	*Central Process Unit*
CSCW	*Computer-Supported Cooperative Work*
DBMS	*DataBase Manager System*
DTD	*Document Type Definition*
EDI	*Eletronic Data Interchange*
EDS	Editor Dirigido por Sintaxe
GUI	*Graphical User Interface*
HTML	*Hypertext Markup Language*
HTTP	*Hypertext Transfer Protocol*
IDL	*Interface Definition Language*
JDK	*Java Development Kit*
JIT	*Just In Time*
JNI	*Java Native Interface*
JSDK	*Java Standard Development Kit*
JVM	*Java Virtual Machine*
LDE	Linguagem de Especificação
MVC	*Model-View-Controller*
NSAPI	*Netscape System API*
OMA	*Object Management Architeture*
OMG	*Object Management Group*

OMS	*Object Management System*
OMT	*Object Modeling Technique*
ORB	*Object Request Broker*
PC	*Personal Computer*
PCTE	*Portable Common Tool Enviroment*
RMI	*Remote Method Invocation*
SGML	*Standard Generalized Markup Language*
TCP	*Transmission Control Protocol*
TCP/IP	*Transmission Control Protocol / Internet Protocol*
UDP	*User Datagram Protocol*
UIMS	*User Interface Manager System*
UML	*Unified Modeling Language*
URL	*Uniform Resource Locator*
W3C	*World Wide Web Consortium*
XML	*Extended Markup Language*

Lista de Figuras

FIGURA 2.1 – Ciclo de transformações em um EDS. 22
FIGURA 2.2 – Exemplo de um documento XML [BRO 2000]. 27
FIGURA 2.3 – Programas existentes no processamento de banco de dados [KRO 99]. 30
FIGURA 2.4 – Programas para aplicações Cliente-Servidor de banco de dados [KRO 99]. 31
FIGURA 2.5 – Funções dos computadores cliente e servidor [KRO 99]. 32
FIGURA 2.6 - Diferença entre bibliotecas de classes e *framework* [PRE 95]. 35
FIGURA 2.7 - Projeto orientado a pontos adaptáveis proposto por [PRE 95]. 36
FIGURA 2.8 – Modelo de Interação MVC. 38
FIGURA 3.1 – Estrutura da arquitetura MVC [GAM 95] 43
FIGURA 3.2 – Estrutura do Padrão *Composite* [GAM 95]. 43
FIGURA 3.3 – Exemplo de composição de objetos [GAM 95]. 44
FIGURA 3.4 – Visão lógica do padrão *Thin Web Client* [CON 2000]. 46
FIGURA 3.5 – Visão lógica do padrão *Thick Web Client* [CON 2000]. 47
FIGURA 3.6 – Visão lógica do padrão *Web Delivery* [CON 2000]. 49
FIGURA 3.7 – Um pedido cliente/servidor usando ORB. 53
FIGURA 3.8 – Ciclo de execução de um servidor *servlet* [DAV 98]. 54
FIGURA 3.9 – Estágios de execução de um *servlet* [DAV 98]. 56
FIGURA 3.10 – Camadas do sistema RMI [RMI 97]. 58
FIGURA 3.11 – Esquema de classes para I/O *stream* em arquivos [CHA 98]. 60
FIGURA 4.1 – Estrutura de um sistema colaborativo [FAR 98]. 71
FIGURA 4.2 – Modelo MVC implementado para o FreDoc. 80
FIGURA 4.3 – Pacotes da arquitetura proposta para o desenvolvimento de Editores Distribuídos de Geração Colaborativa de Documentos. 81
FIGURA 4.4 – Diagrama de Classes do Pacote Classes de Edição. 81
FIGURA 4.5 – Diagrama de Classes do Pacote Formas Geométricas Básicas. 82
FIGURA 4.6 – Diagrama de Classes do Pacote Classes de Controle de Acesso. 83
FIGURA 4.7 – Diagrama de Classes do Pacote Editor Distribuído. 84
FIGURA 4.8 – Modelo de Classes da Arquitetura para Editores Distribuídos de Manipulação Colaborativa de Documentos Diagramáticos. 85
FIGURA 4.9 – Mecanismos de controle de edição de objetos gráficos do FreDoc. 89
FIGURA 5.1 – Hierarquia de classes implementada para o Editor Distribuído de Manipulação Colaborativa de Documentos para diagramas de classes da UML 93

FIGURA 5.2 – Diagrama de classes do pacote *shapes* criado para apresentação gráfica dos objetos durante o processo de edição. 94
FIGURA 5.3 – Página inicial de acesso ao Editor Distribuído de Manipulação Colaborativa de Documentos Diagramáticos. 96
FIGURA 5.4 – Página de cadastro usuários. 97
FIGURA 5.5 – Página de agradecimento ao usuário visitante. 97
FIGURA 5.6 – Página para *Login* do usuário para o Editor Distribuído de Manipulação Colaborativa de Documentos Diagramáticos. 98
FIGURA 5.7 – Menu Principal do FreDocUML. 99
FIGURA 5.8 – Tela de edição de diagramas de classes (UML) disponível para esta versão do FreDocUML. 100
FIGURA 5.9 – Especificação de atributos e métodos para um objeto do tipo classe para o diagrama em edição. 101
FIGURA 5.10 – Geração de documentos XML com as definições do diagrama de classes editado no FreDocUML. 102
FIGURA 5.11 – Colaboração através da inserção de sugestões na especificação de um objeto do tipo classe. 103
FIGURA 5.12 – Salvando um documento diagramático editado pelo FreDocUML. 104
FIGURA 5.13 – Abrindo um documento diagramático editado pelo FreDocUML. 105
FIGURA 5.14 – Ferramenta auxiliar para envio de *e-mail* do FreDocUML. 106

Lista de Tabelas

TABELA 4.1 – Taxinomia Espaço Temporal.. 70

Resumo

Este trabalho apresenta um *framework* para editores distribuídos de manipulação colaborativa de documentos diagramáticos (*FreDoc*). *FreDoc* fornece um conjunto de classes que propicia um ambiente distribuído para edição colaborativa de diagramas de duas dimensões. Ele é constituído de 4 pacotes de classes: 1) de controle de acesso; 2) de controle de edição; 3) formas geométricas básicas; e 4) editor distribuído.

Além disso, é apresentado um estudo sobre mecanismos para construção de editores distribuídos. Editores distribuídos são ferramentas úteis no processo de criação de documentos em um ambiente colaborativo tal como a *World Wide Web*. O uso de editores distribuídos torna o processo de colaboração mais rápido, pois o colaborador participa ativamente do processo de construção através da inserção de anotações que. Estas anotações podem ser aceitas ou rejeitadas total ou parcialmente pelo ou coordenador.

Por fim, é descrito um protótipo, o FreDocUML, desenvolvido para testar a aplicação do *framewrok* FreDoc, mostrando um processo colaborativo de edição de diagrama de classes na notação UML é descrito.

Palavras-chave: Engenharia de *Software*, sistemas distribuídos, editores colaborativos, colaboração.

TITLE: *"DISTRIBUTED EDITOR FOR COLLABORATIVE MANIPULATION OF DIAGRAMMATIC DOCUMENTS"*

Abstract

This work presents a framework for distributed editors for collaborative manipulation of diagrammatic documents (FreDoc). FreDoc supplies a set of classes that produces a collaborative environment for edition of diagrams of two dimensions. It is comprised of 4 packages of class: 1) of access control; 2) of control of editing; 3) basic geometric forms; and 4) distributed editor.

A survey on mechanisms for construction of distributed editors is presented. Distributed editors useful tools in the process of document creation in a collaborative environment such as the World Wide Web. The use of distributed editors makes the process of collaboration much faster, a collaborator participates actively in the building process through the insertion of annotations. Those annotations can be accepted or rejected totally or partially by the author or coordinator.

Finally, FreDocUML, a prototype developed to test the applicability of the FreDoc framework, showing a collaborative process of edition of class-diagrams in UML is described.

Keywords: *Software Engineering, distributed systems, collaborative editor, collaboration.*

1 Introdução

Pode-se definir uma aplicação colaborativa como um sistema computacional que apóia pessoas que estejam desenvolvendo uma tarefa em comum, considerando cada uma delas trabalhando em seu próprio computador, porém compartilhando dados através de interfaces distribuídas, independentes de plataforma. O principal objetivo das aplicações colaborativas consiste em fornecer aos usuários ferramentas que lhes permitam coordenar suas atividades de trabalho.

Khoshafian [KHO 95], por exemplo, cita aplicações que facilitam a interação a distância entre membros de uma equipe de trabalho, tais como: 1) aplicações de comunicação como *chats*, voltadas ao fornecimento de um canal através do qual mensagens podem ser enviadas corretamente; 2) aplicações de colaboração, nas quais grupos de usuários trabalham colaborativamente visando completar uma tarefa; e 3) aplicações para tomada de decisão coletiva (GDSS - *Group Decision Support System*), que suportam o trabalho de um grupo de pessoas visando tomar decisões mediante a apresentação de alternativas. Esta última diferencia-se das aplicações de comunicação, pois suporta a geração de idéias e seleção de alternativas, e das aplicações colaborativas, porque o resultado gerado pelo grupo pode não ser a combinação direta de trabalhos individuais. Apesar das diferenças existentes entre estas aplicações, todas apresentam um fator comum que consiste em facilitar o processo de compartilhamento de informações entre grupos de pessoas que realizam tarefas em comum.

A construção de sistemas colaborativos, segundo Jim Farley [FAR 98], é vista como uma tarefa complexa envolvendo diversas áreas de conhecimento tais como: sistemas distribuídos, comunicações (local ou remota), *interface*, banco de dados, etc. Arquiteturas compostas de tipos distintos de *software*, tais como: *browsers*, gerenciadores de banco de dados, gerenciadores de documentos, *applets*, servidores de aplicação, sistemas e componentes construídos através de linguagens que utilizam técnicas de orientação a objeto (Java, por exemplo) e estruturadas em camadas, geralmente são utilizadas no desenvolvimento de ferramentas colaborativas. Durante este processo são realizadas diversas atividades que utilizam ferramentas distintas e notações dependentes umas das outras, mas que atuam de forma independente.

Como resultado deste processo, tem-se a manipulação e troca de diversas informações geradas nas etapas de desenvolvimento de sistemas. Estas informações vistas como documentos, e a modelagem realizada em um ambiente distribuído, tornam possível a geração de *software* que pode ser visto como um sistema de gestão de documentos distribuídos, possibilitando a colaboração entre grupos de usuários. Para Tadeu Cruz [CRU 98] *groupware* é qualquer sistema computadorizado que permita que grupos de usuários trabalhem de forma colaborativa a fim de atingir um objetivo comum.

O principal objetivo de *groupware* é dar assistência aos grupos na comunicação, colaboração e coordenação de suas atividades. O termo *groupware* já tinha sido usado na década de 80, mas passou a ser adotado pela comunidade CSCW (*Computer-Supported Cooperative Work*) para definir as tecnologias comerciais que procuram implementar sistemas CSCW [BOR 95]. Denomina-se de *Groupware* a tecnologia que aborda as áreas de colaboração, interação homem-máquina e interação homem-homem através do computadores. Existem várias categorias de *groupware* definidas como: comunicação (por exemplo, *e-mail*, comunicação em grupo), colaboração (por exemplo, edição de documentos em grupo) e coordenação (por exemplo, gerência de escalonamento) [KHO 95].

Outro ponto importante a ser tratado em aplicações colaborativas é como está implementado o processo de colaboração entre usuários. A colaboração através de anotações é uma forma de comunicar idéias ou opiniões sobre o conteúdo de um documento [SAP 2001]. Um usuário colaborador analisa o documento recebido, insere suas anotações e as "envia" ao usuário proprietário do documento colaborado. Estas anotações são inseridas com o intuito de aperfeiçoar o documento, e também podem ser utilizadas como registro das argumentações ocorridas durante o processo de criação do mesmo. Além disso, as anotações servem como um mecanismo de comunicação entre o usuário proprietário e os usuários colaboradores. As anotações podem ser utilizadas para apresentar idéias ou opiniões (através da inserção de comentários), ou para sugerir alterações no documento (através de anotações de substituição, inclusão ou remoção de um componente). Este processo de colaboração já é empregado em ferramentas como editor de texto Word 97 da Microsoft®, *Rational Rose* [RAT 96] e o EDI [POM 99].

O problema concentra-se no desenvolvimento de editores específicos para um determinado tipo de documento. Segundo Esperança [ESP 93], o desenvolvimento de um editor

específico para uma determinada linguagem é um processo que demanda muito tempo e esforço. Além disso, os editores diferenciam-se basicamente por suas respectivas linguagens, sendo que as demais características de tais editores apresentam um alto grau de semelhança. Pode-se pensar então na construção de geradores de editores, devendo estes reunir desde componentes comuns associados até instância de componentes dependentes de linguagem. Assim, os esforços na criação e manutenção de editores específicos seriam bastante reduzidos. Existe ainda a possibilidade de torná-los acessíveis em ambientes distribuídos, através da definição de componentes comuns para controlar a colaboração.

No entanto, a reutilização de código pode ser feita através de mecanismos fornecidos pelo paradigma de orientação a objetos, possibilitando o reuso de uma aplicação através da interface (permitindo acesso aos serviços já implementados) ou através da redefinição do comportamento de algumas subclasses. Assim, pode-se obter diferentes aplicações utilizando como base uma aplicação já existente, reutilizando tanto o código como o projeto geral dessa aplicação [CAM 97]. Segundo Silva [SIL 2000], *framework* é um esqueleto de implementação de uma aplicação ou de um subsistema de aplicação, em um domínio de problema particular. É composto de classes abstratas e concretas e provê um modelo de interação ou colaboração entre as instâncias de classes definidas pelo *framework*. Um *framework* é utilizado através de configuração ou conexão de classes concretas e derivação de novas classes concretas a partir das classes abstratas do *framework*.

Um *framework* é considerado como uma estrutura de classes que fornece o comportamento necessário para implementar aplicações para um dado domínio, através de mecanismos existentes em linguagens orientadas a objetos [GAM 95].

Basicamente, um *framework* é composto de dois níveis. Um nível mais abrangente que fornece a estrutura de controle da aplicação, constituída pelas classes do *framework*, e um nível inferior, constituído por subclasses concretas implementadas pelo projetista. O nível inferior permite a implementação de operações específicas cujas descrições são modeladas no nível genérico. As operações podem especializar a estrutura de controle de acordo com os requisitos da aplicação específica, bem como chamar operações definidas no nível genérico [DEU 89], [CAM 97] e [SIL 96].

1.1 Proposta

A criação de editores de documentos para ambientes distribuídos apresenta uma vantagem: a colaboração. Porém, há obstáculos que dificultam o processo de criação destes editores. A dificuldade de especificação de tais editores está relacionada com a inflexibilidade dos componentes (de edição e visualização) e armazenamento de documentos, levando à criação de estruturas dados pré-definidas de um determinado tipo de documento. Isso cria a necessidade de se especificar um conjunto de classes (projetista do editor) para armazenamento dos dados de um documento e, posteriormente, em como estes serão utilizados em ambientes distribuídos, como no EDI [POM 99], por exemplo, onde os dados modelados são temporariamente armazenados em estruturas de dados pré-definidas para posterior utilização pelo seu ambiente distribuído de edição gráfica.

Considerando as vantagens de editores textuais/diagramáticos no processo de geração de documentos [FAV 88], [ESP 93], [MEL 89], [MAR 89a] e [MAR 89b], bem como a necessidade de se implementar colaboração entre usuários através de ambientes distribuídos, a criação de um *framework* que sirva como base para o desenvolvimento de editores distribuídos na *web* (visando a criação e colaboração de documentos) pode ser uma solução para o problema proposto.

Neste sentido, a proposta deste trabalho é o desenvolvimento de um *framework* a ser utilizado como base na criação de ferramentas gráficas para geração colaborativa de documentos diagramáticos, através de *interfaces* GUI (*Graphical User Interface*) disponíveis via *web*.

Para alcançar este objeto realizou-se, inicialmente, um estudo sobre ferramentas já existentes, seguido da definição de um *framework,* o FreDoc, que propicia a geração colaborativa de documentos XML, em um ambiente distribuído e colaborativo. Neste trabalho foi explorada a funcionalidade provida por XML para criação de documentos a partir de informações armazenadas em banco de dados, disponível através de uma classe Java que transforma documentos diagramáticos em documentos XML. FreDoc foi desenvolvido com o objetivo de apoiar interações assíncronas de profissionais envolvidos na edição colaborativa de documentos diagramáticos. Seu desenvolvimento foi motivado pela necessidade de apoiar diversos profissionais geograficamente dispersos na tarefa de edição de documentos.

O FreDoc foi construído tendo como base o EDI [POM 99] e o HotDraw for Java [KNO 97], ambos ambientes que disponibilizam edição colaborativa de documentos diagramáticos.

Através de analises realizadas sobre o HotDraw for Java [KNO 97] e o EDI [POM 99] foi possível definir a estrutura para a base do FreDoc, visto que estes disponibilizam um conjunto de classes para edição de objetos gráficos pela *web*. Destas ferramentas, foram aproveitadas suas classes base de controle e atualização de visões do objetos gráficos em edição. Optou-se, porém, na separação das informações textuais e gráficas dos objetos utilizados na representação de documentos diagramáticos em edição, armazenando-as em estruturas de dados dinâmicas que, posteriormente, são enviadas para *servlets* Java, responsáveis enviá-las para armazenamento e recuperá-las de banco de dados implementados em um servidor *web*. Utilizou-se, também, a serialização de objetos para possibilitar o envio de informações, entre o cliente e o servidor, em forma de objetos de classes persistentes utilizadas na edição de objetos gráficos.

Como resultado tem-se a validação do FreDoc através da implementação de um protótipo, o FreDocUML. Seu principal objetivo é fornecer suporte para a edição de diagramas de classes, baseados em UML [BOO 99], [LAR 99] e [ERI 98]. Diagramas gerados por este editor são visualizados pelos usuários e, através dos recursos de edição disponíveis, é implementada a colaboração. Os documentos editados são estruturados segundo o *framework* MVC (*Model-View-Controller*) [BUS 96] ,[PAR 94], [BAR 95a] e [BAR 95b]. Finalmente, os documentos gerados compõem documentos XML, visando portabilidade para outras ferramentas que interpretam XML.

1.2 Organização do trabalho

O trabalho está dividido em 6 capítulos e anexo, que estão descritos a seguir.

O capítulo 2 apresenta características de tipos de editores, juntamente com o estudo de mecanismos para compartilhamento de arquivos. Este capítulo também abordado o conceito de *framework* e a utilização deste para definir a estrutura de ferramentas de editores gráficos. O objetivo deste capítulo é descrever os recursos que deverão compor editores que serão gerados para servir de suporte ao processo de geração colaborativa de documentos. Neste trabalho o

conceito de *framework* será empregado na construção de uma arquitetura que servirá de base para o desenvolvimento do FreDoc, um *framework* para desenvolvimento de instâncias de editores diagramáticos em ambientes distribuídos.

No capítulo 3 estão descritos os padrões utilizados na construção de ferramentas de edição textual ou gráfica, juntamente com padrões que podem ser utilizados para especificar arquiteturas de aplicações que serão projetadas para *web*. Estes padrões foram utilizados no FreDoc para auxiliar na definição e implementação de classes de controle e edição de documentos e também para auxiliar na construção e distribuição de classes de acesso e armazenamento de informações sobre documentos diagramados pelo FreDocUML, protótipo implementado para validar FreDoc.

O capítulo 4 trata da definição do *framework* para desenvolvimento de editores distribuídos de manipulação colaborativa de documentos diagramáticos, o FreDoc. Apresenta-se também, o mecanismo de colaboração a ser utilizado por este *framework*.

O protótipo desenvolvido com base no FreDoc, o FreDocUML, que propicia um ambiente gráfico de edição para modelagem de diagrama de classes da UML com suporte à colaboração, é apresentado no capítulo 5. O FreDocUML implementa características descritas nos capítulos anteriores, utilizadas como base para seu desenvolvimento.

As conclusões estão apresentadas no capítulo 6, juntamente com propostas para continuidade de desenvolvimento do protótipo utilizando as especificações do FreDoc.

Os anexos apresentam as classes *FrameAccessScreen* e *SimpleDrawingCanvas*, e os diagramas de casos de uso e de seqüência do *framework* FreDoc.

2 Editores e mecanismos de compartilhamento de documentos

Neste capítulo serão apresentadas as características de alguns dos tipos de editores, juntamente com alguns mecanismos de compartilhamento de documentos que poderão ser utilizados como suporte para a colaboração entre usuários, durante o processo de edição de um determinado documento. Apresenta-se também o conceito de *framework* e como estes podem ser utilizados na confecção de ambientes de edição de documentos.

2.1 Editores Textuais e Diagramáticos

2.1.1 Editores Dirigidos por Sintaxe

Segundo [FAV 88], [ESP 93] e [MAC 89], um Editor Dirigido por Sintaxe (EDS) é um editor de textos programável. A sintaxe e semântica da linguagem a ser processada pelo EDS são descritas em Linguagem de Especificação (LDE). A LDE permite a especificação sintática e semântica de uma linguagem, através de uma gramática de atributos estendida por variáveis globais e ações semânticas. O texto em edição é armazenado internamente na forma de uma Árvore Sintática abstrata. Um módulo do sistema cria uma descrição interna da linguagem a partir da especificação feita em LDE. O EDS e as demais ferramentas que ficam acopladas a ele utilizam constantemente a tabela do reconhecedor (criada internamente por um módulo do sistema) na edição, submissão e formatação do texto.

Um EDS pode ser definido como uma ferramenta interativa para a edição de textos que se diferencia dos editores tradicionais por possuir conhecimento da sintaxe e da semântica estática de uma linguagem.

A partir do conhecimento da estrutura sintática dos documentos, os editores dirigidos por sintaxe fornecem facilidades de edição que compreendem formatação automática, auxílios à visualização e auxílios à submissão de texto. O conhecimento da semântica estática da linguagem permite a verificação dos aspectos dependentes de contexto.

Segundo [ESP 93], editores dirigidos por sintaxe são editores de texto orientados à edição de linguagens específicas, onde os documentos são visualizados e alterados em função de

estruturas sintáticas das linguagens. EDSs oferecem também facilidades como formatação automática e verificação da semântica estática dos documentos.

Em editores de texto convencionais, os textos são representados internamente como uma seqüência de linhas, onde cada linha é composta por uma seqüência de caracteres. Em um EDS, a representação interna constitui-se em uma estrutura de árvore sintática[1].

Todas as operações de edição são definidas em função de um foco de atenção, que corresponde à posição corrente do cursor. O foco de atenção em editores convencionais está associado a um caracter de uma linha. Em um EDS, o foco de atenção, associado à representação da árvore sintática, indica um nodo da árvore sintática.

As operações de inclusão, alteração ou remoção de texto em um EDS são realizadas sobre a estrutura interna do documento. Após cada uma destas operações – que modificam a árvore sintática – é apresentado ao usuário o texto correspondente à nova árvore sintática.

Em conformidade com [FAV 88], o funcionamento básico de um EDS pode ser resumido como um ciclo de transformações da representação concreta para a abstrata e transformações da abstrata para a concreta, como ilustrado na figura 2.1. O processo de submissão de texto (*parse*) compreende a transformação da representação concreta (texto) para a representação abstrata (árvore sintática). O processo de formatação (*unparse*) compreende a transformação da representação abstrata (árvore sintática) para a concreta (texto).

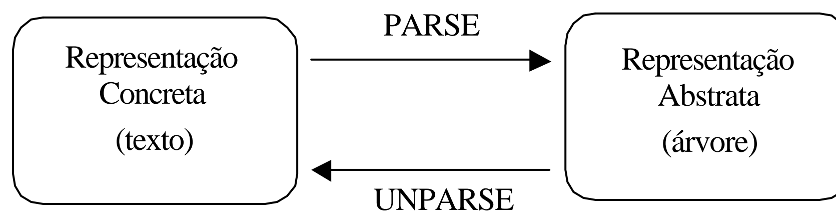


FIGURA 2.1 – Ciclo de transformações em um EDS.

O primeiro sistema que utilizou a abordagem estruturada foi o sistema *Emily* [HAN 71]. O sistema *Menthor* [DON 84] criou uma coleção de ferramentas específicas para processar árvores sintáticas abstratas para a linguagem Pascal. Gandalf [HAB 86] explorou formas de

[1] Uma Árvore Sintática representa o caminho de derivação percorrido no reconhecimento de uma sentença. Seu nodo raiz está associado ao símbolo inicial da gramática, seus nodos internos estão associados aos símbolos não-terminais e os nodos folha representam os símbolos terminais.

integrar editores de programas para linguagens específicas com facilidades de execução e depuração.

Com a evolução deste tipo de ferramenta, surgiram os geradores de editores interativos orientados a uma linguagem. *Synthesizer Generator* [REP 88] e *Programming System Generator* [BAH 86] são os exemplos mais representativos de geradores de editores dirigidos por sintaxe. O problema destes tipos de editores é o desempenho, pois o reconhecedor tem que ficar processando as árvores sintáticas abstratas e normalmente são interpretados.

2.1.2 Editores Diagramáticos

Segundo [MEL 89], [MAR 89a], [MAR 89b] e [THE 87], diagramas são ferramentas essenciais na construção de programas e sistemas complexos. Abaixo são citadas e comentadas algumas funções desempenhadas pelas técnicas estruturadas baseadas em notações diagramáticas, e benefícios que se espera alcançar com o desenvolvimento de editores diagramáticos:

Clareza de raciocínio: Através do uso de técnicas diagramáticas estruturadas, o editor de documentos diagramáticos tem a sua disposição uma poderosa ferramenta de auxílio ao raciocínio, pois poderá ver e expressar um pensamento com maior clareza. A utilização de notações diagramáticas adequadas pode aumentar a rapidez e a qualidade dos trabalhos realizados [NEU 90].

Comunicação entre os membros participantes do processo de colaboração: Documentos diagramáticos, em geral, são editados por um grupo de pessoas. Neste caso diagramas são ferramentas essenciais de comunicação. O uso de técnicas diagramáticas é bastante útil no sentido de facilitar troca de idéias entre os colaboradores de forma padronizada e, também, auxiliar o acoplamento de componentes desenvolvidos separadamente [NEU 90].

Facilidade de manutenção: Diagramas bem estruturados podem ser de grande ajuda aos engenheiros de *software* nas tarefas de manutenção. Os diagramas os auxiliam a descobrir como os programas foram desenvolvidos e projetados, permitindo-lhes, assim, entender e prever melhor os efeitos colaterais ocasionados devido às atualizações realizadas [NEU 90].

Facilidade de depuração: Diagramas bem estruturados são valiosas ferramentas de ajuda à depuração erros (de programas, por exemplo), pois auxiliam os encarregados desta tarefa

a entender melhor os requisitos e o projeto do programa. Neste sentido, os diagramas facilitam a compreensão dos objetivos e funcionamento dos programas, facilitando a delimitação da área a ser depurada [NEU 90].

Facilidade na geração de documentação: Diagramas bem estruturados são muito importantes como ferramentas de documentação. Eles podem ser utilizados para definir as especificações e para representar o projeto. Eles permitem que as descrições arquiteturais e detalhadas dos programas sejam feitas de maneira fácil e consistente [NEU 90].

Apesar destas vantagens, diagramas são extremamente difíceis de se criar e manter manualmente. Os editores diagramáticos foram justamente desenvolvidos para suportar interfaces gráficas com as quais colaboradores interagem desenhando diagramas. Editores diagramáticos possibilitam que diagramas possam ser armazenados e recuperados facilmente [MEL 89].

2.2 Mecanismos de Compartilhamento

2.2.1 Eletronic Data Interchange

2.2.1.1 Características básicas

EDI (*Eletronic Data Interchange*) é definido como a troca de dados entre sistemas heterogêneos para suportar transações [GOC 98], compreendendo além da exportação de dados de um sistema para outro, também a interação entre os sistemas. Para [COL 2000] EDI é um processo de troca de dados em formato eletrônico entre aplicações e/ou plataformas heterogêneas de forma que eles possam ser processados sem intervenção manual. Mensagens EDI possuem conjuntos de elementos de dados que podem ser obrigatórios, opcionais ou condicionais. Tais conjuntos formam documentos de negócios (*business documents*) que são transmitidos entre parceiros caracterizando transações EDI [GOC 98]. Segundo [DIS 2000] a sigla EDI é utilizada para denominar todo e qualquer processo que realize troca de dados através de um meio eletrônico entre sistemas heterogêneos.

O EDI surgiu para simplificar, automatizar e agilizar todos os processos que envolvem a comunicação entre as organizações. Os benefícios de sua implantação incluem

economia de papel, fax e telefone, melhorando as relações entre os parceiros comerciais. Ao disponibilizar de maneira rápida e precisa as informações facilita-se também o planejamento das atividades diárias das empresas [COL 2000].

Apesar de sua larga utilização em comércio eletrônico, as regras definidas para EDI geram alguns efeitos colaterais indesejáveis. Com a evolução de diversas tecnologias, principalmente relacionadas à Internet, os problemas com EDI têm se tornado cada vez mais intoleráveis, o que tem levado empresas a procurar novas formas de implementação de intercâmbio de documentos. As principais desvantagens da utilização de EDI na forma tradicional são: (1) exigência de uma solução única; (2) formato muito rígido; (3) evolução de padrões muito lenta; (4) altos custos fixos; (5) necessidade de integração total entre sistemas; (6) ausência de um padrão claro; (7) falta de suporte para definição de conteúdo dinâmico [BRO 2000].

EDI foca-se demais no processo como uma parte integral do conjunto de transações. Novas tecnologias, como XML, suportam a separação do processo e regras de negócio do conteúdo e estrutura de dados.

2.2.1.2 Evolução dos documentos

A partir de 1970 começaram a ser introduzidos os primeiros sistemas de automação de processos de negócios. EDI foi introduzido pelo TDCC (*Transportation Data Coordinating Committee*). O TDCC criou conjuntos de transações para fornecedores de modo a permitir o processamento eletrônico de ordens de compra e faturas [GOC 98]. EDI se refere ao uso de conjuntos rígidos de transações com regras de negócios embutidas neles [GOC 98]. Há várias especificações de padrões associadas a EDI. As mais comuns são EDIFACT (*EDI for Administration, Commerce and Transport*), usada primeiramente nos círculos internacionais, e ANSI X.12 (ANSI – *American Nacional Standards Institute*) [CIN 99].

Entretanto, EDI possui problemas que limitam seu crescimento. Um dos mais significativos é o fato de estar baseado na transferência de conjuntos fixos de transações. Esta rigidez torna difícil tratar a evolução necessária às empresas para introduzir novos produtos ou serviços, ou desenvolver ou substituir sistemas computacionais. Além disso, o alto custo fixo de implementação de EDI não é justificável para empresas de pequeno e médio porte [COL 2000].

Para Charles [GOC 98], a solução pode ser a construção de uma nova arquitetura de EDI associadas a tecnologias de XML (*Extensible Markup Language*), internet, serviços baseados na internet e conectividade de banco de dados.

2.3 XML (*Extensible Markup Language*)

XML (*Extensible Markup Language*) [W3C 98] é um subconjunto da SGML (*Standard Generalized Markup Language*) [ARB 95] definido no padrão ISO 8879:1986, otimizado para utilização na *Internet*. Os elementos que compõem o documento podem ser descritos numa estrutura formal denominada *Document Type Definition* (DTD). Através do DTD, é possível realizar a validação de documentos XML. Ao contrário do SGML, a existência de um DTD não é obrigatória, sendo que o XML atribui uma definição padrão para componentes de marcação não declarados. XML é flexível, de forma que permite descrever qualquer estrutura lógica de texto, tais como memorandos, cartas, ofícios, etc. XML não é um conjunto pré-definido de *tags*, como HTML (*Hypertext Markup Language*).

Seu processo de padronização é gerenciado pelo W3C (*World Wide Web Consortium*). O principal objetivo de XML é facilitar a troca de documentos estruturados através da Internet. Tal objetivo aliado à forma de implementação escolhida para atingi-lo, faz com que XML seja um forte candidato à integração com EDI para a formação de editores que possibilitem a intercâmbio de documentos. Isso acontece devido ao fato de XML ser uma notação universal com um formato de dados que mantém o conteúdo e a estrutura, mas separa as regas de negócio dos dados. Como resultado, para cada tipo de editor de documento pode-se aplicar suas próprias regras de negócio, eliminando os problemas de EDI descritos anteriormente [GOC 98].

XML elimina as opções sintáticas de SGML, mas mantém a possibilidade da descrição dos dados através de estruturas abstratas utilizando um vocabulário próprio [CAS 99]. Sem que a idéia de "marcação estruturada" do SGML fosse deixada de lado, todas as características complexas e raramente utilizadas foram removidas. Apesar de cortes radicais na definição de SGML, XML ainda é compatível. XML é "um SGML *lite*" [MAC 97]. XML separa totalmente os dados propriamente ditos das instruções de apresentação do documento.

Um documento XML normalmente é constituído por três partes principais, sendo as duas primeiras opcionais [BRY 97]:

- uma identificação da versão de XML utilizada, o conjunto de caracteres utilizado pelo documento e se o documento faz referências a outros arquivos;
- uma declaração de tipo de documento que indica o DTD pelo qual o documento deve ser validado;
- a instância do documento (figura 2.2) que contém de fato o conteúdo do documento, organizado na forma de uma hierarquia de elementos. O conteúdo do documento é chamado dessa forma porque, caso o documento apresente um DTD, seu conteúdo pode ser visto como uma instância da classe de documentos definida pelo DTD.

A declaração da versão XML somente é opcional para permitir que documentos SGML e HTML possam ser utilizados como documento XML. A declaração da versão do XML utilizado no documento permite que futuras versões XML consigam tratá-lo da maneira adequada. A declaração do tipo do documento constitui a base do conceito de validação estrutural, que torna aplicações baseadas em XML robustas e confiáveis. O DTD é a formalização da idéia intuitiva de tipo de documento. O DTD lista os tipos de elementos disponíveis e pode inserir restrições na ocorrência e conteúdo dos elementos, além de outros detalhes da estrutura dos elementos. Isto torna o sistema de informação mais robusto por forçar a consistência dos documentos que dele fazem parte.

```
<Documento_ID = “Modelagem.XML”>
        <Nome Classe=”Cliente”>
            <Atributo>
                    <Nome>ClienteID</Nome>
                    <Tipo>Integer</Tipo>
            </Atributo>
            <Atributo>
                    <Nome>NomeCliente</Nome>
                    <Tipo>String</Tipo>
            </Atributo>
        </Nome Classe>
        <Nome Classe>
            ...
        </Nome Classe>
<Documento_ID>
```

FIGURA 2.2 – Exemplo de um documento XML [BRO 2000].

2.3.1 Principais características da linguagem XML

Inteligência: XML é inteligente para qualquer nível de complexidade. A marcação pode ser alterada de uma marcação mais geral como "<PESSOA> Paulo </PESSOA>" para uma mais detalhista, como "<PESSOA> <AGENDA> <AMIGO> Paulo </AMIGO> </AGENDA> </PESSOA>". As idéias são bem marcadas para que "<PROJETO> aluno </PROJETO>" e "<CLASSE> aluno </CLASSE>" sejam sempre valores diferentes. A informação conhece a si mesma [BRO 2000].

Adaptação: XML possui uma adaptação infinita. Marcações personalizadas podem ser criadas para atender qualquer necessidade [BRO 2000];

Manutenção: XML é fácil de manter. Ela contém somente idéias e marcações. Folhas de estilos e *links* vêm em separado, e não escondidas no documento. Cada um pode ser alterado separadamente quando preciso, com fácil acesso e fáceis mudanças [BRO 2000];

Ligação: XML é capaz de realizar ligações não suportadas por HTML. HTML pode fazer ligações simples, onde um objeto liga-se a outro. XML faz isso, mas também pode ligar dois ou mais pontos a uma idéia. Existem ainda *links* gêmeos que ligam todas as idéias dentro de uma mesma. Qualquer *link* entre uma idéia pode ser manipulado de uma única maneira [BRO 2000];

Simplicidade: XML é simples, se comparada com a SGML. A especificação de SGML tem 300 páginas. A de XML, 33. Idéias obscuras e desnecessárias foram retiradas em favor de idéias concisas [BRO 2000];

Portabilidade: XML é de fácil portabilidade. XML pode ser navegada com ou sem o seu DTD (*Document Type Definition*, ou Definição de Tipo de Documento - as normas que definem como as *tags* são estruturas nos documentos XML), tornando o *download* mais rápido. Tudo que um navegador precisa para ver XML é ter a noção de que ele próprio e a folha de estilos controlam a aparência. Se uma validação estrita é necessária, o seu DTD pode acompanhá-la e fornecer detalhes exatos da sua marcação [BRO 2000].

2.3.2 XML: Objetivos do desenvolvimento

A especificação da XML primou pelos seguintes objetivos [CON 97]:

- tornar clara a utilização de XML na Internet;
- suportar uma grande variedade de aplicações;
- ser compatível com SGML;
- facilitar a escrita de programas que processem documentos XML;
- manter em um mínimo absoluto (ideal de zero) o número de recursos opcionais em XML;
- tornar os documentos legíveis pelos seres humanos e razoavelmente claros;
- tornar o projeto XML rápido, formal e conciso;
- facilitar a criação de documentos;
- tornar irrelevante a concisão na marcação.

Como enfatiza [CON 97], a *Web* está se transformando numa espécie de inteligência "*cyborg*": onde homens e máquinas trabalham juntos para gerar e manipular informação.

HTML é muito limitada principalmente por não poder ser extensível à representação dos mais diversos tipos de informação, capaz apenas de marcação estrutural e não semântica. Por outro lado, SGML é muito complexa para ser facilmente implementada em navegadores que precisam manter-se como aplicações leves para serem amplamente utilizados em plataformas mais desprovidas em termos de CPU e memória.

Assim, XML parece ser um bom compromisso entre a flexibilidade em termos de representação informacional e a simplicidade necessária para tornar-se uma ferramenta ubíqua na *web* [CON 97].

A passagem da marcação estrutural de HTML para a marcação semântica de XML é uma fase crítica no esforço para transformar a *web* de um espaço global de informação em uma rede universal de conhecimento.

2.4 Banco de dados cliente-servidor

Um banco de dados cliente-servidor é um sistema no qual pelo menos um servidor armazena e processa um banco de dados [KRO 99]. Para melhor entender o processo cliente-servidor, apresenta-se na figura 2.3 os programas existentes no processamento de um banco de dados. Os programas de aplicação processam a interface do usuário, empregam a lógica da aplicação e impõem algumas regras de negócio (restrições). Os programas de aplicação chamam o SGBD para os serviços de banco de dados (para banco de dados relacionais, tais solicitações de serviços são expressas geralmente em alguma forma de SQL).

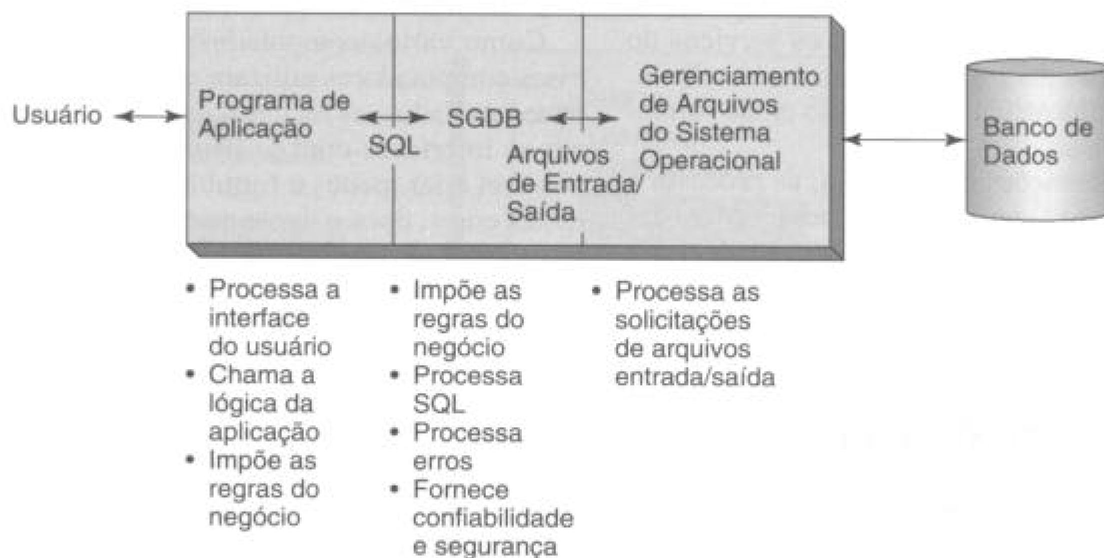


FIGURA 2.3 – Programas existentes no processamento de banco de dados [KRO 99].

O SGBD processa as instruções SQL e pode impor regras de negócios ao mesmo tempo. Ele retorna dados e mensagem de erro para o programa de aplicação. Além disso, o SGBD fornece as funções de confiabilidade e segurança. Finalmente, o SGBD envia as

solicitações de entrada/saída para o sistema operacional. Segundo David Kroenke [KRO 99], todas estas funções são obrigatórias em qualquer sistema de banco de dados, independentemente de seu tipo.

2.4.1 Funções dos computadores cliente e servidor

Para David Kroenke [KRO 99], em sistemas cliente-servidor, o programa de aplicação é colocado nos computadores cliente, e o SGBD e os programas de gerenciamento de arquivos dos sistemas operacionais são colocados no computador servidor, como mostra a figura 2.4. Observe que uma parte do SGBD, chamado *driver do SGBD*, e colocado no cliente.

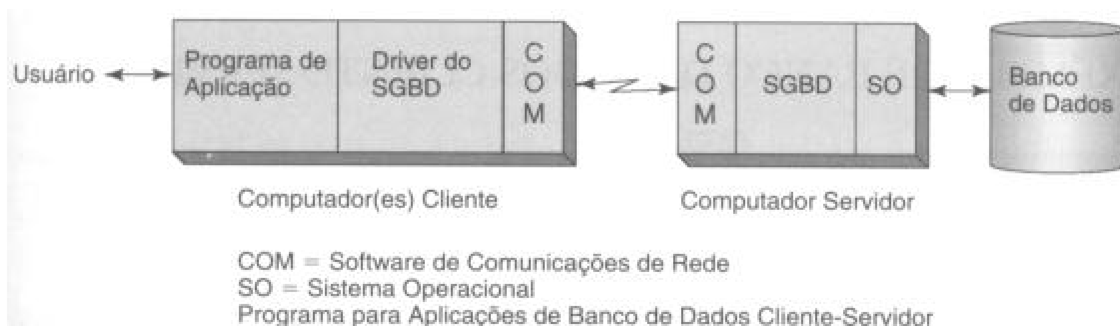


FIGURA 2.4 – Programas para aplicações Cliente-Servidor de banco de dados [KRO 99].

A função do driver do SGBD é receber as solicitações de processamento de aplicação e formatá-las para enviá-las ao SGBD. Ele também recebe resposta do SGBD e as formata para a aplicação. Uma camada de *software* de comunicações é colocada tanto no cliente como no servidor. A função desse *software* é interagir com o *hardware* de comunicação para enviar e receber mensagens entre o *driver* do SGBD e o próprio SGBD. Já a figura 2.5 resume as funções do cliente e do servidor. O computador cliente gerencia a interface com o usuário, aceitando seus dados, processando a lógica da aplicação, impondo as regras de negócio e gerando solicitações para os serviços do banco de dados. A seguir, os clientes transmitem essas solicitações ao servidor e recebem os resultados, que são então formatados para o usuário.

O servidor aceita as solicitações dos clientes, as processa e então envia uma resposta. Ao fazer isso, o servidor administra as regras de negócio, realiza a verificação de integridade, faz

a manutenção dos dados de *overhead* do banco de dados e possibilita o controle sobre acesso concorrente. O servidor também executa os serviços de segurança e recuperação.

Funções do Cliente	Funções do Servidor
Gerenciar a interface do usuário.	Aceitar dos clientes as solicitações ao banco de dados.
	Processar as solicitações ao banco de dados.
Processar a lógica da aplicação.	Formatar os resultados e transmiti-los ao cliente.
Impor as regras do negócio.	Impor as regras do negócio.
Gerar solicitações ao banco de dados (SQL).	Realizar a verificação de integridade.
Transmitir para o servidor as solicitações ao banco de dados.	Manter os dados de overhead do banco de dados.
Receber os resultados do servidor.	Fornecer controle do acesso concorrente.
Formatar os resultados.	Fornecer serviços de recuperação e segurança.

FIGURA 2.5 – Funções dos computadores cliente e servidor [KRO 99].

2.5 *Frameworks*

Segundo [JOH 92], um *framework* é constituído por um conjunto de classes abstratas e componentes que, em conjunto, definem um projeto abstrato para uma família de problemas relacionados. [GAM 95], por sua vez, define que um *framework* é um conjunto de classes cooperantes que constituem um projeto reutilizável para uma classe específica de *software*.

Um *framework* define a arquitetura de um domínio de aplicação. Ele define a estrutura global da aplicação, a sua divisão em classes e objetos, as responsabilidades chave de cada parte, como as classes e os objetos que colaboram e a seqüência de controle. Ambos os autores citados acima destacam aspectos abstratos do projeto como o aspecto central que caracteriza os *frameworks*.

Outro autor, [PRE 94], ressalta os aspectos estruturais de classes e de sua utilização por especialização para a construção de aplicações. Segundo o autor, um *framework* é constituído por um conjunto de blocos de construção prontos para serem utilizados e outros semi-acabados. A arquitetura global, isto é, a composição e interação de blocos, são predefinidas também. Produzir uma aplicação específica usualmente envolve a adaptação de componentes a necessidades específicas implementando alguns métodos em subclasses das classes do *framework*.

Goldberg [GOL 95], por sua vez, centra-se nos aspectos operacionais, ou dinâmicos de um *framework*, tomando em consideração o aspecto da estrutura de controle que esse *framework* fornece para a construção de aplicações. Segundo ele, um *framework* é um conjunto de objetos que interagem e que fornecem um conjunto bem definido de serviços. É uma forma bem definida na qual o controle é transferido entre esses objetos.

Por último, [DEU 89] define o conceito de *framework* levando em consideração o contexto de utilização das abstrações e de extensão dessas abstrações. Para ele, um *framework* pode prover uma instância X de uma classe própria como parâmetro a uma instância Y de uma outra classe existente, com a expectativa de que Y enviará um conjunto de mensagens preestabelecidas à X. Neste caso Y é considerado como o *framework* enquanto X é considerado um cliente interno para diferenciá-lo dos clientes externos que interagem com a combinação. Um cliente pode definir uma subclasse XC de uma classe existente YC, fornecendo funcionalidade adicional ou especializada. Outra vez, a classe existente é considerada com o *framework* e o cliente em cliente interno.

Estas definições enfatizam diferentes aspectos ou perspectivas, tanto da estrutura como da natureza operacional dos *framework*. A definição de Deutsch [DEU 89], é talvez a mais precisa em termos dos mecanismos que a orientação a objetos oferece para suportar reutilização de funcionalidade. Esta definição, entretanto, não destaca os aspectos de *abstração* inerentes aos

frameworks, nem os aspectos de domínio de aplicação, fazendo referência implícita à predefinição de uma estrutura de controle. Este aspecto é ressaltado pela definição de Goldberg [GOL 95], mas, neste caso, o aspecto de abstração está implícito na definição. Pree, e também, Gamma, Helm, Johnson e Vlissides [PRE 94], incluem o conceito de arquitetura como um dos aspectos essenciais que caracterizam um *framework*, o qual está intimamente relacionado com a noção de estrutura de controle bem definida referida por Goldberg [GOL 95].

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

Desta forma, pode-se dizer que um *framework* é um conjunto de classes concretas e abstratas e a interface entre elas tem por objetivo servir como uma aplicação semi-acabada, utilizada como base para o desenvolvimento de novas aplicações. Aplicações baseadas em um *framework* são construídas customizando suas classes. Segundo Pree [PRE 95], reusar um *framework* significa adaptar sua estrutura para necessidades específicas, sobrescrevendo métodos de algumas classes nas subclasses.

Para [DEU 89], a reutilização de elementos de *software* pode ocorrer ao nível de código, ou de projeto, sendo esta a forma mais importante. A reutilização de código consiste na utilização direta de trechos de código já desenvolvidos. A reutilização de projeto consiste no reaproveitamento de concepções arquitetônicas de uma aplicação em outras, não necessariamente com a utilização da mesma implementação [SIL 96].

As classes de um *framework* tornam seu projeto genérico, definindo protocolos de comunicação que utilizam métodos do tipo abstrato, *template* e *hook*. Desta forma, os métodos podem ser classificados de acordo com seu comportamento em: Métodos base, Métodos abstratos, Métodos *hook* e Métodos *template*.

Um *framework* é um projeto genérico em um domínio que pode ser adaptado a aplicações específicas, servindo como um molde para a construção de aplicações. Um *framework* pode fornecer um auto nível de reutilização, pois além de fornecer reutilização de código, oferecido por classes e objetos, também oferece reutilização de projeto. A figura 2.6 ilustra esta

característica através de apresentação do fluxo de controle de um programa construído com ou sem a utilização de um *framework*. Se um *framework* não é utilizado, então o programador necessita conhecer os momentos onde os métodos pertencentes às bibliotecas de classes devem ser chamados. Diferentemente, através da utilização de um *framework* o programador não precisa saber quando chamar tais métodos, pois eles pertencem à estrutura interna do *framework*. O programa do usuário limita-se a definir as novas funcionalidades que são desejadas, deixando a cargo do *framework* decidir quando chamar tais funcionalidades.

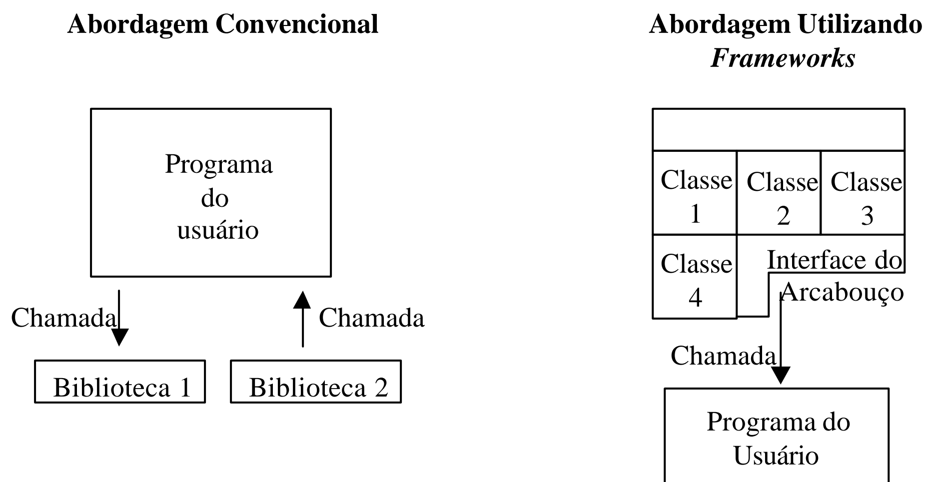


FIGURA 2.6 - Diferença entre bibliotecas de classes e *framework* [PRE 95].

2.5.1 Desenvolvimento de *Frameworks*

Segundo [PRE 95], a construção de um *framework* utilizando-se da abordagem de projeto orientado a pontos adaptáveis consiste na identificação dos pontos fixos (*frozen spots*) e dos pontos adaptáveis (*hot spots*) do domínio de uma aplicação. Os pontos fixos correspondem às partes comuns das aplicações correlatas, enquanto que os pontos adaptáveis são as partes que podem ser estendidas para cada aplicação específica, dando ao *framework* a capacidade de ser flexível e moldar-se a diferentes aplicações. Esta flexibilidade é obtida através da utilização dos metapadrões. A figura 2.7 ilustra as etapas do projeto dirigido a pontos adaptáveis.

A etapa inicial consiste na identificação e definição da estrutura de classes do *framework*. Esta etapa requer a utilização de uma metodologia de desenvolvimento de *software*.

A segunda etapa consiste na identificação dos pontos adaptáveis. É preciso identificar os aspectos que variam de aplicação para aplicação, assim como o grau de flexibilidade desejado.

Reprojetar um *framework* consiste na modificação da estrutura definida inicialmente de modo a obter a flexibilidade desejada. [PRE 95] propõe a utilização de metapadrões nesta etapa. Entretanto, optou-se pela utilização também dos padrões de projeto sugeridas por [GAM 95].

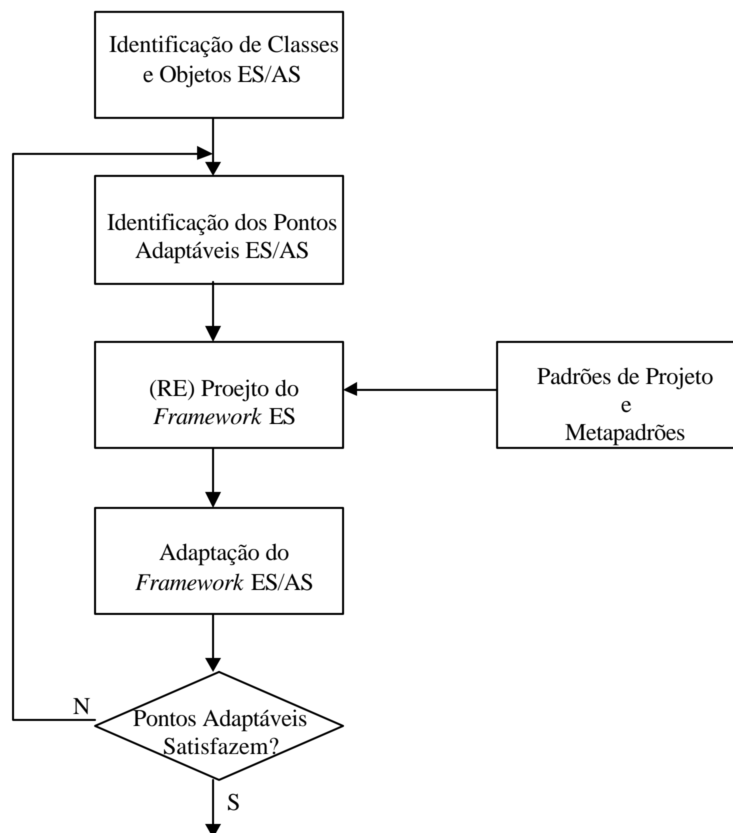


FIGURA 2.7 - Projeto orientado a pontos adaptáveis propostos por [PRE 95].

A etapa final consiste em um refinamento da estrutura do *framework*. Ao final, o *framework* é avaliado para verificar se os pontos adaptáveis oferecem o grau de flexibilidade desejado. Caso isto não ocorra, retorna-se à etapa de identificação de pontos adaptáveis.

A respeito da descrição de *frameworks*, Johnson [JOH 97] afirma que "*frameworks*, eliminam a necessidade de uma nova notação de projeto através do uso de uma linguagem de programação orientada a objetos como notação de projeto".

2.5.2 *Frameworks* no processo de construção de editores diagramáticos

Ferramentas desenvolvidas para *frameworks* específicos constituem um outro tipo de ferramenta para assistir o desenvolvimento de aplicações baseadas em *frameworks*. A ferramenta de *canvas* do VisualWorks [PAR 94] e o construtor de ferramentas de HotDraw [BRA 95] constituem de dois exemplos destas ferramentas. A ferramenta de *canvas* de VisualWorks permite a construção de interfaces gráficas através da seleção de elementos visuais em um *palette* e sua alocação em um *canvas*. O produto resultante é uma descrição de interface em forma declarativa, a ser interpretada por um objeto construtor de interfaces, em uma aplicação baseada em MVC. O construtor de ferramenta de HotDraw é um *browser* que facilita a tarefa de associação de *figures*, *commands* e *readers* (instâncias de dasses do *framework*) para definir a estrutura de ferramentas em editores gráficos. Estas ferramentas auxiliam apenas o uso de seus respectivos *frameworks*. Segundo [SIL 2000], ferramentas para *frameworks* específicos, o que inclui a ferramenta de *canvas* de VisualWorks e o construtor de ferramentas de HotDraw, ilustram que alguns *frameworks* podem usar ferramentas específicas, que não são aplicáveis para outros *frameworks*.

2.5.3 Modelo de Interação MVC

Um dos blocos de construção que um *framework* de interface normalmente utiliza é o modelo de interação MVC (*Model–View–Controller*). Sua primeira aparição foi no ambiente de programação *SmallTalk*–80 [PAR 94]. MVC foi projetada para reduzir o esforço de programação necessário na implementação de sistema que apresentam vários dados de forma sincronizada. O principal objetivo da arquitetura MVC é a separação da interação homem-máquina do domínio do problema e do modo como isto é gerenciado [BRA95a] e [BRA95b]. Esta separação (dados e aplicação) gera maior flexibilidade e grande possibilidade de reuso.

O modelo MVC utiliza três classes conforme a figura 2.8. A classe *Model* gerência os dados de domínio específico que serão representados e manipulados pela aplicação GUI. A classe *View* representa todos ou alguns destes dados na tela. A classe *Controller* é responsável por

aceitar entrada (*mouse* e teclado) e por passar as mensagens apropriadas às classes *Model* e *View* para habilitar a edição do modelo de dados.

Por exemplo, se a aplicação for um processador de textos: o *Model* armazena os dados textuais, o *View* representa o texto numa janela de edição e o *Controller* gerencia a digitação. Já em uma aplicação gráfica o *Model* armazena informações sobre objetos gráficos, o *View* representa a figura numa janela de edição e o *Controller* gerencia a edição destas figuras.

Uma aplicação é criada a partir de um *framework* MVC gerando-se subclasses. Especializações nas subclasses promovem o refinamento das classes MVC para habilitar a visualização e edição dos dados do modelo. Visões e Controladores devem ter apenas um Modelo, mas Modelos podem ter várias Visões e Controladores.

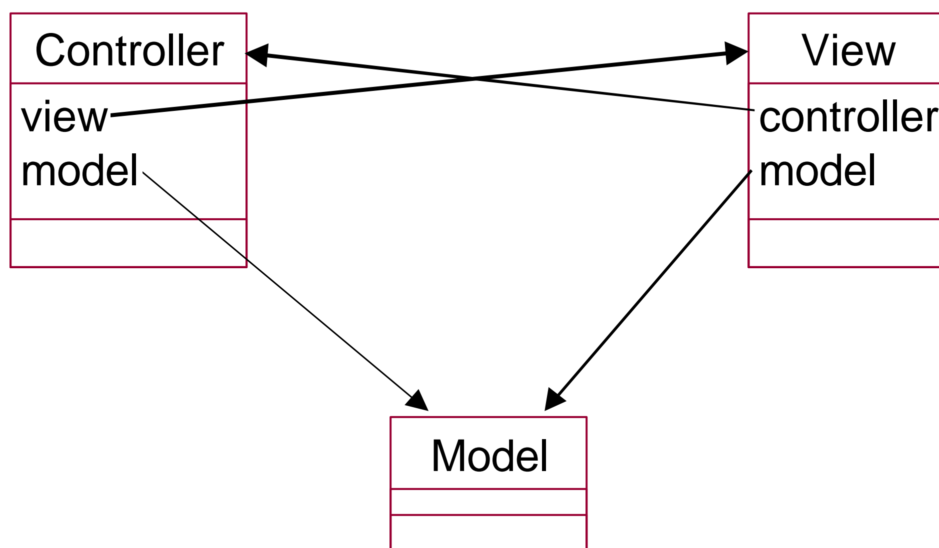


FIGURA 2.8 – Modelo de Interação MVC.

2.5.3.1 Atualizações MVC

Visões e controladores são geralmente amarrados. A razão torna-se óbvia se for considerada a diferença entre editar dados numa planilha ou num aplicativo de diagramas gráficos. Os tipos de interações envolvidas em uma visão de planilha são completamente diferentes das interações envolvidas numa visão gráfica.

A possibilidade de múltiplas visões introduz o problema de manter todas as visões consistentes com o estado dos dados, quando editados através de uma das visões. Isto é chamado *problema de atualização MVC*, o qual é usualmente gerenciado mantendo-se uma lista de todas as visões. Assim, uma atualização pode ser propagada por todas as visões.

Pode-se ver a atualização MVC na figura 2.8 mostrada anteriormente. Cada vez que uma visão muda, uma mensagem de mudança é propagada para todas as outras visões através do modelo. Esta aparente simples idéia torna-se muito útil e poderosa.

2.5.3.2 Modelo

O Modelo contém dados de domínio-específico que são exibidos e manipulados por uma aplicação. Eles podem ser uma faixa de inteiros (representando contadores e termômetros), *arrays* de caracteres (editores de texto simples), listas dinâmicas, registros ou outras estruturas de dados complexas.

2.5.3.3 Visão

Uma visão controla a representação visual de todo ou partes de um modelo específico. Funções comuns como *refresh* ou *scroll* de uma janela podem ser mantidas nesta classe, mas funções específicas das aplicações como "exibir um *array* como uma planilha" serão implementadas nas subclasses, pelos desenvolvedores da aplicação.

Visões podem representar todo o modelo ou somente certos aspectos. A visão deve saber sobre o modelo que está representando, mas não precisa de conhecimento sobre outras visões.

2.5.4 Controlador

Controladores são associados tanto a visões quanto a modelos. Um controlador aceita entradas do usuário através de vários dispositivos como teclado e *mouse* e envia mensagens apropriadas ao modelo e visão.

Controladores devem saber sobre o modelo e visão que estão associados, mas não precisam de informação sobre outros controladores.

Comunicações entre o *Model* e o par *View-Controller* são capturadas nas classes abstratas. Assim a arquitetura MVC pode ser reutilizada para novas visões e aplicações. Isto pode economizar um considerável esforço de projeto cada vez que o *framework* MVC e utilizado.

Visões e controladores são associados com seus modelos quando são criados. Cada vez que os dados do modelo são modificados, este envia uma mensagem *broadcast* de mudança para todos os seus dependentes. Cada visão e controlador dependentes podem acessar os dados do modelo e atualizar a si mesmos apropriadamente. Parâmetros passados com a mensagem de mudança permitem às visões e controladores decidir se necessitam de atualização.

2.5.4.1 Arquitetura MVC (*Model-View-Controller*)

No MVC (*Model/View/Controller*) [GAM 95] o modelo (*Model*) contém a funcionalidade e os dados. A Visão (*View*) mostra os dados ao usuário. O Controlador (*Controller*) manipula dados de entrada, como ilustrado na figura 2.10.

Segundo Gamma [GAM 95], o primeiro, e talvez o melhor, exemplo para o padrão *Observer*, aparece em *Smalltalk Model/View/Controller* (MVC), que usa um *framework* para o ambiente de interface. Já para Buschmann [BUS 96] o padrão arquitetural MVC divide uma aplicação interativa em três componentes.

O padrão *Observer* define uma relação de dependência de objetos de um para muitos, de maneira que quando um objeto muda seu estado, todos seus dependentes são notificados e atualizados automaticamente. Este padrão define, basicamente, o funcionamento da arquitetura MVC (*Model-View-Controller*) [PAR 94] e [BRA 95a].

Podem existir múltiplas visões de um mesmo modelo, uma vez que as visões e os controladores estão separados da representação do modelo. A cada visão, é associado um controlador. Se alguma visão altera (através do controlador) o estado do modelo, este notifica todas as outras visões de que seus dados foram modificados.

3 Padrões de projeto e de arquiteturas de aplicações *web* e mecanismos para construção de ambientes distribuídos e heterogêneos

Neste capítulo serão apresentados os padrões utilizados na construção de ferramentas de edição textual ou gráfica juntamente com padrões que podem ser utilizados para especificar arquitetura de aplicações que serão projetadas para *web*.

3.1 Padrões de projeto

Segundo Gamma [GAM 95], padrões de projeto são descrições de objetos e classes comunicantes que são customizados para resolver um problema geral de projeto num contexto particular.

3.1.1 Padrão *Observer*

No MVC (*Model/View/Controller*) [GAM 95] o modelo (*Model*) contém a funcionalidade e os dados. A Visão (*View*) mostra os dados ao usuário. O Controlador (*Controller*) manipula dados de entrada, como ilustrado na figura 3.1. Este padrão pode ser utilizado em aplicações para web na confecção de ferramentas que auxiliem na edição de documentos diagramáticos, pois disponibiliza mecanismos de atualização de visões durante um processo de edição.

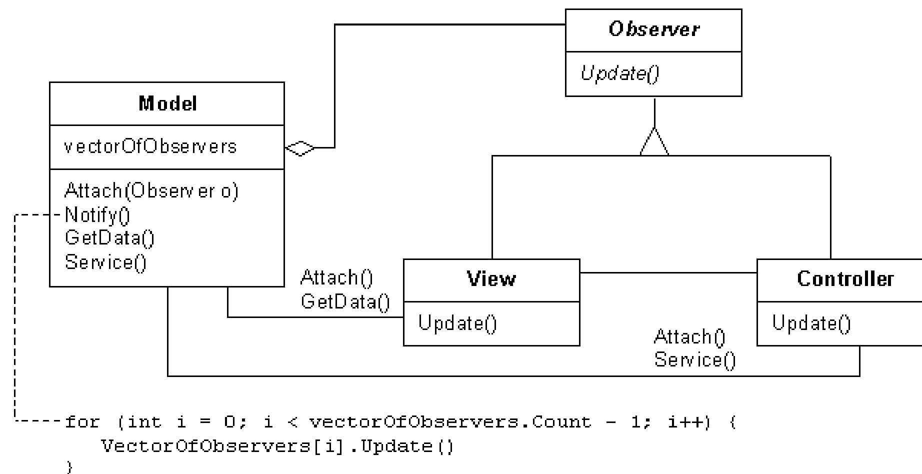


FIGURA 3.1 – Estrutura da arquitetura MVC [GAM 95]

3.1.2 Padrão *Composite*

O padrão *Composite* [GAM 95], é utilizado para compor vários objetos em estruturas hierárquicas. Tanto os objetos individuais como os objetos compostos são tratados de forma idêntica, como mostra a figura 3.2.

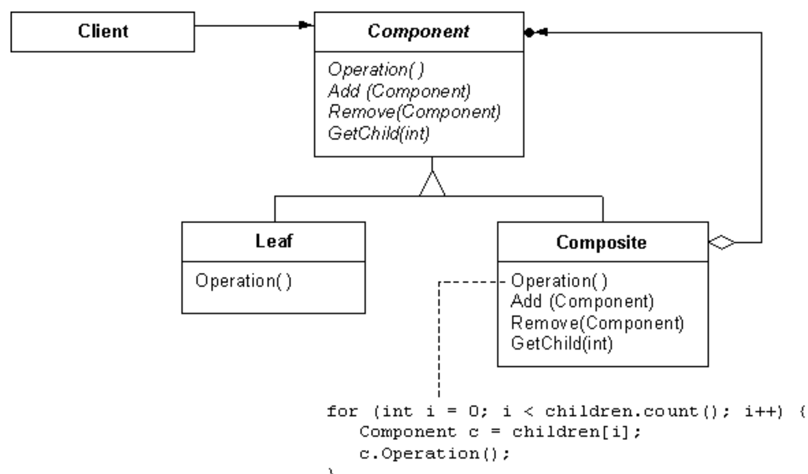


FIGURA 3.2 – Estrutura do Padrão *Composite* [GAM 95].

A classe abstrata *Component* declara uma interface abstrata para os objetos na composição, definindo o comportamento padrão para a interface comum a todas as classes. Da mesma forma define o acesso a seus componentes filhos, e opcionalmente pode definir o acesso ao componente pai.

O objeto *Leaf* representa a folha dentro da árvore hierárquica, não possuindo filhos, definindo um comportamento para objetos primitivos. Já o objeto *Composite* define o comportamento do objeto que possui descendentes, armazenando-os, como ilustrado na figura 3.3. As operações destes objetos é o conjunto de operações executadas pelos filhos.

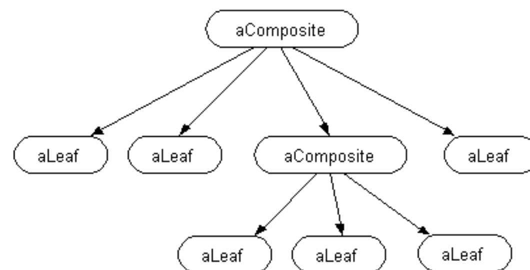


FIGURA 3.3 – Exemplo de composição de objetos [GAM 95].

O padrão torna o cliente simples, uma vez que ele não precisa saber se está tratando com um componente primitivo (*Leaf*) ou composto (*Composite*), tratando-os uniformemente. Facilmente novos objetos podem ser criados – através da composição de outros objetos existentes – sem que o código do cliente precise ser modificado. Através deste padrão, pode-se desenvolver ferramentas que auxiliem editores no processo de edição de documentos diagramáticos, uma vez que sua estrutura possibilita a utilização de objetos básicos, geralmente definidos por uma linguagem de programação, na construção de outros mais complexos, através de composição.

3.2 Padrões de arquitetura para aplicações *web*

Segundo Jim Conallen [CON 2000], as arquiteturas para aplicações *web* são visões de alto nível de componentes arquiteturalmente significantes no sistema. Um componente neste sentido é uma entidade autocontida com uma interface pública. Componentes arquiteturalmente significantes são aqueles que aparecem nas visões mais altas de um sistema. Vários componentes ou coleção de componentes podem ser tipicamente agrupados ou empacotados com outros

componentes similares. A natureza específica destes componentes é, obviamente, dependente de sua visão específica.

Aplicações *web* podem ser vistas como um sistema de *software* cliente/servidor, se possuírem no mínimo os seguintes componentes arquiteturalmente significantes: 1) um *browser* HTML/XML em um ou mais clientes que se comunicam com o servidor *web* via HTTP; e 2) um servidor de aplicação que gerência os serviços.

Um padrão de arquitetura expressa um esquema de organização estrutural fundamental para sistema de *software*. Ele fornece um conjunto pré-definido de sub-sistemas, especifica suas responsabilidades, e inclui regras e normas para organização de relacionamento.

Para Jim Conallen [CON 2000], os três padrões de arquitetura para aplicações *web* mais comuns são: 1) *Thin Web Client* que se utilizadas capacidades mais básicas do *browser*, onde a maioria das transações é executada no servidor; 2) *Thick Web Client* que permite que transações sejam executadas no cliente através de *scripts*, *applets* e controles *ActiveX*, onde uma quantia significante de transações é executada no cliente; e 3) *Web Delivery* onde o cliente participa no sistema de objeto distribuído. Nesta arquitetura, o cliente comunica-se diretamente com outros servidores de objetos, sem a necessidade do uso do HTTP.

3.2.1 *Thin Web Client*

Este padrão é mais apropriado para aplicação *web* baseada na internet ou para aqueles ambientes em que o cliente tem pouco poder computacional ou nenhum controle sobre sua configuração. Os principais componentes deste padrão de arquitetura encontram-se no servidor. São eles: 1) *Client browser*; 2) *Web server*; 3) HTTP *connection*; 4) HTML *page*; 5) *Server page*; e 6) *Application server*. A figura 3.4 mostra um diagrama da visão lógica para a arquitetura *Thin Web Client*. Este padrão de arquitetura poderá sofrer adaptações para atender as especificações da aplicação a ser desenvolvida. Para tanto, deve-se acrescentar novos componentes arquiteturalmente significativos à visão lógica mostrada na figura 3.4.

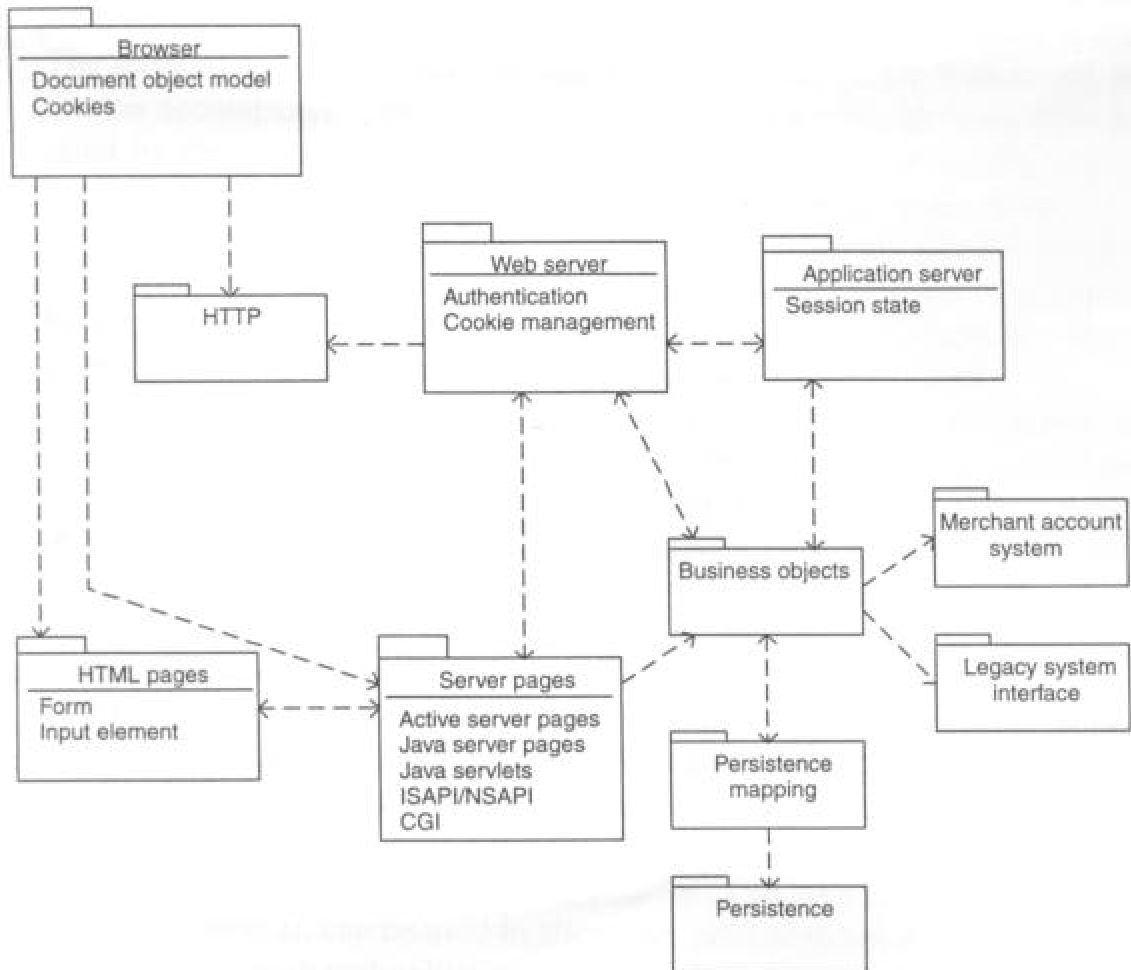


FIGURA 3.4 – Visão lógica do padrão *Thin Web Client* [CON 2000].

3.2.2 *Thick Web Client*

O padrão de arquitetura *Thick Web Client*, mostrado na figura 3.5, estende o padrão *Thin Web Client* com o uso de *scripts* pelo cliente e objetos personalizados, tais como controles *ActiveX* e *applets* Java, permitindo ao cliente executar algumas das transações do sistema. As duas maiores motivações para o uso deste padrão são a melhoria da interface com o usuário e a execução de transações pelo cliente. Por ser uma extensão do padrão *Thin Web Client*, o padrão de arquitetura *Thick Web Client* apresenta os mesmos componentes, acrescentado dos seguintes elementos: 1) *Client script*; 2) XML *document*; 3) *ActiveX control*; 4) Java *applet*; e 5) *JavaBean*. A figura 3.5 mostra um diagrama da visão lógica para a arquitetura *Thick Web Client*.

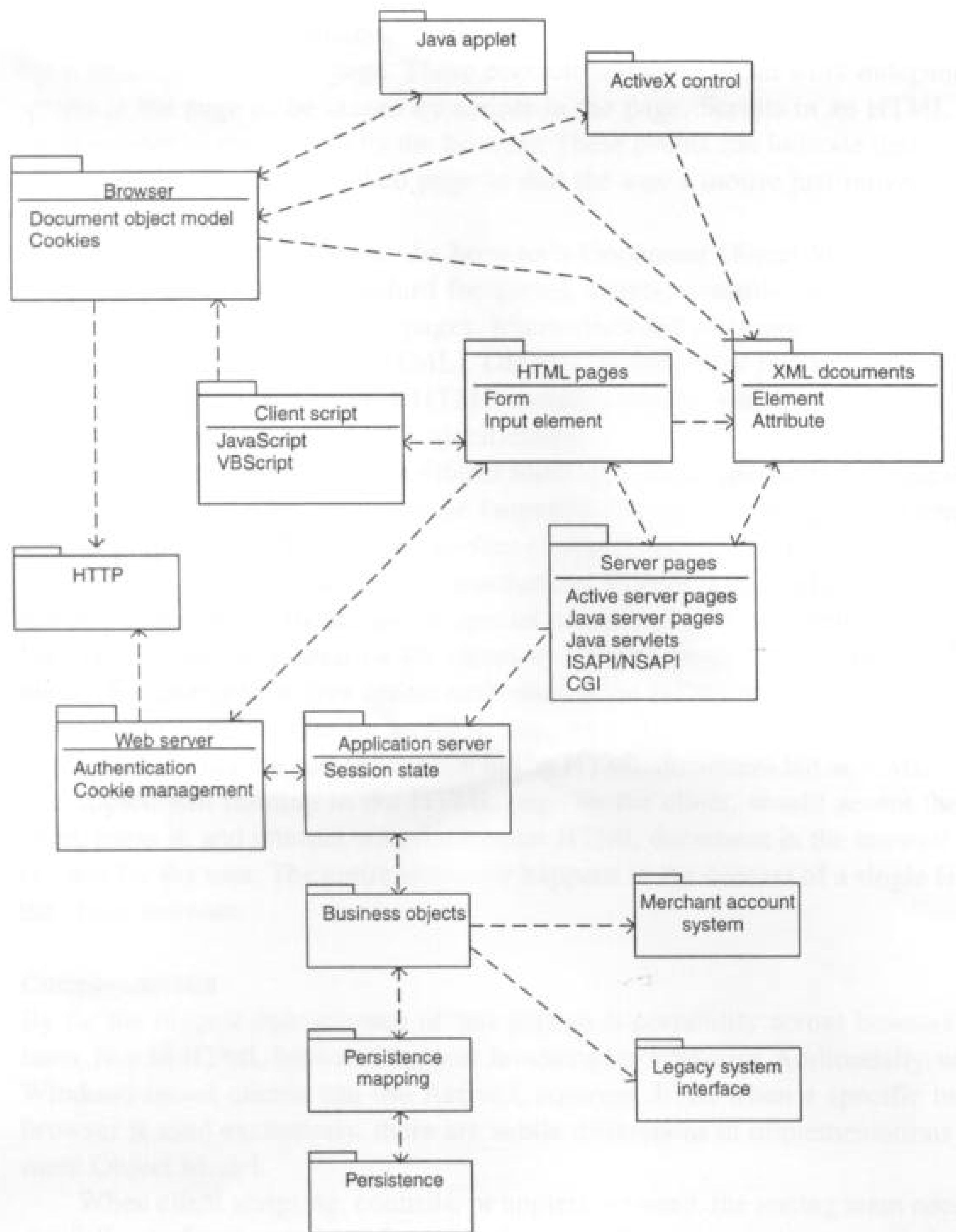


FIGURA 3.5 – Visão lógica do padrão *Thick Web Client* [CON 2000].

3.2.3 *Web Delivery*

O padrão de arquitetura *Web Delivery*, é assim chamado por que a *web* é utilizada como um mecanismo despachante para um sistema tradicional cliente/servidor de objeto distribuído. O padrão de arquitetura *Web Delivery* apresenta os mesmos componentes do padrão *Thin Web Client*, além de utilizar outros recursos de comunicação, tais como DCOM, IIOP e RMI, garantindo interação entre o cliente e o servidor. A figura 3.6 mostra um diagrama da visão lógica para a arquitetura *Web Delivery*.

3.3 Mecanismos para Construção de Ambientes Distribuídos e Heterogêneos

Nesta seção serão apresentados alguns recursos que suportam a distribuição de objetos em ambientes heterogêneos. Serão apresentadas também tecnologias utilizadas para o desenvolvimento de aplicações distribuídas tais como CORBA e *Servlet*.

3.3.1 Arquitetura de Aplicações em Camadas

Uma arquitetura em camadas dita o caminho através do qual aplicações serão criadas e como os seus componentes serão distribuídos através do sistema [HAM 99]. A maioria das aplicações é feita de três tipos fundamentais de componentes: um componente de apresentação (que contém a lógica que apresenta a informação a uma fonte externa e obtém entradas daquela fonte); um componente de negócio (que contém a lógica da aplicação que governa as funções de negócios e os processos executados pela aplicação) e; um componente de acesso a dados (que contém a lógica que fornece à interface um sistema de armazenamento de dados ou com algum outro tipo de fonte de dados externos, com uma aplicação externa).

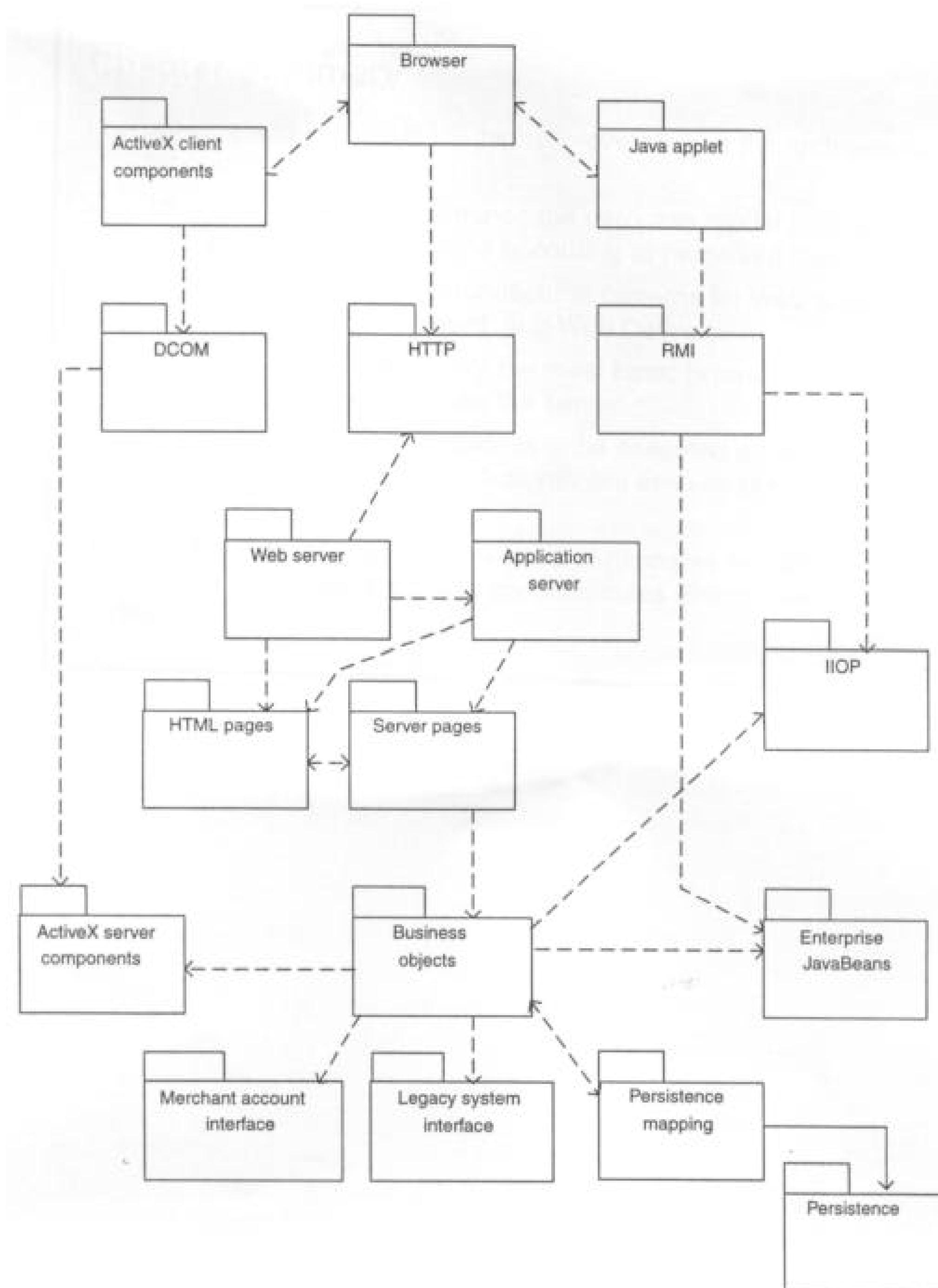


FIGURA 3.6 – Visão lógica do padrão *Web Delivery* [CON 2000].

3.3.1.1 Aplicações *one-tier*

A arquitetura *one-tier* é baseada em um ambiente onde todos os componentes são combinados num simples programa integrado, rodando somente em uma máquina. Essa arquitetura, totalmente centralizada, corresponde ao tradicional ambiente *mainframe* [EDW 97].

A alternativa *one-tier* oferece uma quantidade significativa de vantagens. Sendo a aplicação centralizada em um único ambiente, é fácil gerenciar, controlar e garantir a segurança de acesso a essas aplicações. Esses sistemas são seguros, confiáveis e suportam vários usuários.

3.3.1.2 Aplicações *two-tier*

Com o advento dos computadores pessoais, redes locais, bancos de dados relacionais e poderosas aplicações e ferramentas *desktop*, a indústria de computação direcionou-se para o mundo dos sistemas "abertos" e cliente/servidor [EDW 97]. Tomadores de decisão podem gerar seus próprios relatórios e manipular dados com ferramentas *desktop* poderosas, em suas próprias estações de trabalho. Esse tipo de arquitetura permite a manipulação de funções, anteriormente impossíveis de ser manipuladas.

A arquitetura cliente/servidor em duas camadas divide o processamento entre uma estação *desktop* e uma máquina servidora. O ambiente cliente/servidor mais popular usa um PC (*Windows-based*) com uma poderosa ferramenta de desenvolvimento GUI (*Graphical User Interface*) e um servidor de banco de dados UNIX ou Windows NT. Algumas vantagens surgem dessa arquitetura. As ferramentas GUI possibilitam um maior desempenho no desenvolvimento e distribuição de aplicações. Considerando a existência de um processamento distribuído entre o cliente e o servidor, as máquinas servidoras não necessitam ser tão potentes, o que resulta em custos mais baixos do que os sistemas *mainframe*. Os sistemas de bancos de dados, por independerem de plataforma, permitem portabilidade mais fácil entre sistemas, efetivamente quebrando a dependência no fornecimento do *hardware*. Considerando a facilidade de uso das ferramentas GUI, o nível de qualificação dos desenvolvedores não precisa ser alto [EDW 97].

Em contrapartida, surgem algumas desvantagens: perda de segurança, confiança e controle são algumas delas. Esse modelo é extremamente eficaz para aplicações de médio porte, acessando poucos bancos de dados e não suportando uma grande quantidade de usuários. Sem as facilidades de controle e segurança, disponíveis nos sistemas centralizados, cada aplicação cliente

deve cuidar do seu próprio processo de segurança. Com isso, muitas companhias ainda relutam em mover suas aplicações de missão crítica para PCs [EDW 97].

3.3.1.3 Aplicações *multi-tier*

Estas aplicações conseguem obter o melhor das arquiteturas anteriores, sem as suas desvantagens. Além de suportar os benefícios de ambas as arquiteturas (*one-tier* e *two-tier*), a arquitetura *multi-tier* também suporta os benefícios de uma arquitetura bastante flexível [EDW 97].

As três camadas referem-se às três partes lógicas que compõem uma aplicação, e não ao número de máquinas usadas pela aplicação. O modelo *multi-tier* divide a aplicação nas seguintes camadas: lógica de apresentação, lógica de negócio e lógica de acesso aos dados. Os componentes da aplicação comunicam-se entre si utilizando uma interface abstrata, que funciona como um contrato, em ternos do que está sendo tornado público. Essa mesma camada abstrata esconde todos os detalhes de implementação das funções desempenhadas por um componente. Ela identifica a operação a ser realizada e define os parâmetros de entrada e saída necessários à execução da operação. Esse tipo de infra-estrutura possibilita serviços de localização, segurança e comunicação entre os componentes da aplicação. Neste último tipo de aplicação consegue-se extrair algumas vantagens das aplicações desenvolvidas tais como: reutilização de objetos por outras aplicações, facilidade de manutenção do sistema, independência entre aplicação e banco de dados, dentre outras.

3.3.2 CORBA – *Common Object Request Broker Architecture*

A especificação, reconhecida hoje em dia como padrão de mercado para implementação de aplicações *multi-tier* é a especificação CORBA, que é o mais importante e ambicioso projeto de *middleware* já desenvolvido pela indústria de *software*. Uma das primeiras especificações adaptadas pelo OMG (*Object Management Group*) foi à especificação CORBA. Ela detalha as interfaces e as características do componente ORB (*Object Request Broker*) de uma OMA (*Object Management Architecture*). CORBA especifica um sistema que fornece interoperabilidade entre objetos em um ambiente distribuído heterogêneo e de modo transparente para o programador, sendo seu projeto baseado no objeto modelo do OMG [VIN 98] e [ZHO 99].

O componente central do CORBA é o ORB. Ele circunda toda a infra-estrutura de comunicação necessária para identificar e localizar objetos, manusear gerenciamento de conexão e salva dados. Em geral, o ORB não é requerido para ser um único componente, simplesmente

definindo interfaces. O ORB central é a parte mais crucial do ORB, pois é responsável pela comunicação de pedidos.

Em 1991 foi introduzido o CORBA 1.1 pelo OMG e foi definida uma Linguagem de Definição de Interfaces IDL (*Interface Definition Language*), e interfaces de programação de aplicações API (*Application Programming Interfaces*), que permitem a objetos cliente/servidor interagirem com uma implementação específica de um ORB. CORBA 2.0, adotado em dezembro de 1994, definiu verdadeira interoperabilidade especificando como ORBs de diferentes fabricantes podem interoperar.

O ORB é um *middleware* que estabelece os relacionamentos cliente/servidor entre objetos, conforme ilustrado na figura 3.7. Usando um ORB, um cliente pode transparentemente invocar um método em um objeto servidor, que pode estar na mesma máquina ou na rede. O ORB intercepta a chamada e é responsável por achar um objeto que pode implementar o pedido, passar os parâmetros, invocar seu método e retornar o resultado. O cliente não tem que ser informado de onde o objeto foi localizado, sua linguagem de programação, seu sistema operacional, ou nenhum aspecto do sistema que não seja parte da interface do objeto. Assim, o ORB fornece interoperabilidade entre aplicações de máquinas diferentes em ambientes heterogêneos distribuídos e interconecta múltiplos sistemas objetos.

3.3.2.1 *Object Request Broker* – ORB

Um ORB CORBA 2.0 fornece mais que um básico mecanismo de comunicação utilizando linguagens heterogêneas, ferramentas, plataformas e redes. Ele também fornece um ambiente para manusear objetos, notificando suas presenças e descrevendo seus metadados. Um ORB CORBA é a própria descrição do caminho. Com CORBA 2.0, um ORB pode quebrar interações entre objetos que residem dentro de um único processo (como um programa em C++), bem como entre objetos que globalmente interagem através de múltiplos vendedores ORB e sistemas operacionais.

O ORB CORBA 2.0 é considerado o primeiro de todos os *middlewares* cliente/servidor [VIN 98]. É um *middleware* que estabelece os relacionamentos cliente/servidor entre objetos. Usando um ORB, um objeto cliente pode transparentemente invocar um método em um objeto servidor, que pode estar na mesma máquina ou através da rede. O ORB intercepta a chamada e é responsável por achar o objeto que pode implementar o pedido, passando os parâmetros, invocando o método e retornando o resultado, como ilustrado na figura 3.7. O cliente não tem que estar informado de onde o objeto estava localizado, sua linguagem de programação, seu sistema operacional, ou nenhum outro aspecto que não seja parte da interface do objeto. É muito importante notar que as regras cliente/servidor são usadas apenas para coordenar as interações entre dois objetos. Objetos em um ORB podem ativar seu próprio cliente ou servidor, dependendo da ocasião.

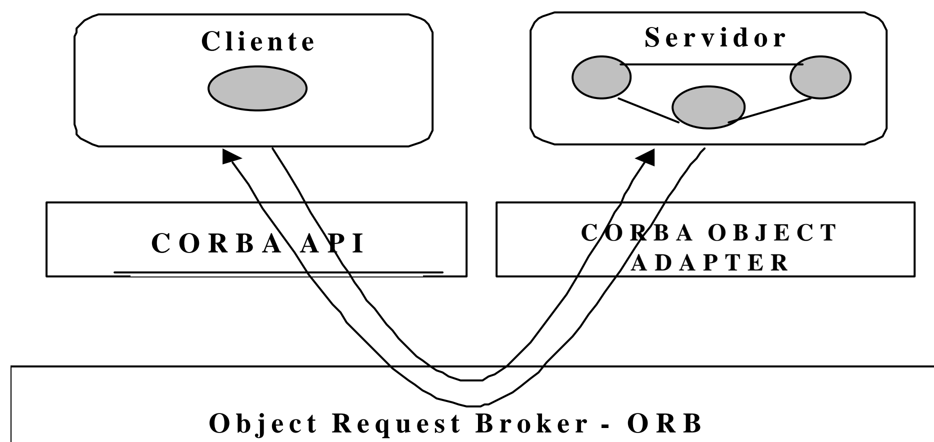


FIGURA 3.7 – Um pedido cliente/servidor usando ORB.

3.3.3 *Servlets*

S*ervlets* são programas em Java utilizados para estender as funcionalidades de um servidor *Web*. *Servlets* são para o servidor enquanto que *applets* são para o cliente [DAV 98], [ALM 2000] e [COL 98].

Um *Servlet* define um conjunto de métodos, sem interface gráfica para usuário, que trabalha juntamente com o *Servlet Engine*, ambiente escrito por uma distribuição de servidor *Web* de acordo com a especificação Java *Servlet* API, sendo executado no servidor *Web* que, por sua vez, trata das requisições e respostas aos clientes [ALM 2000].

O programa cliente é qualquer programa (escrito em Java), capaz de fazer conexões e requisições ao servidor *Web*. Essa requisição é processada pela *Servlet Engine* que está rodando no servidor *Web*. Depois de feita a requisição do cliente para *Servlet Engine*, ela retorna uma resposta a *Servlet*. Em seguida a *Servlet* envia uma resposta dentro do protocolo **HTTP** para o cliente como ilustrado na figura 3.8.

Funcionalmente, *Servlets* estão entre programas **CGI** e extensões servidoras proprietárias **NSAPI**. Exceto em relação ao fato de que, em se tratando de programa escritos em Java, é necessário modificar uma *Servlet* para especificar um tipo de plataforma. Além disso, *Servlets* apresentam outras vantagens como:

- fornecem uma maneira de gerar documentos dinâmicos que são fáceis de escrever e rápidos para executar;
- são mais rápidos que *scripts* **CGI**;
- usam uma **API** padrão que é suportada por muitos servidores *Web*;

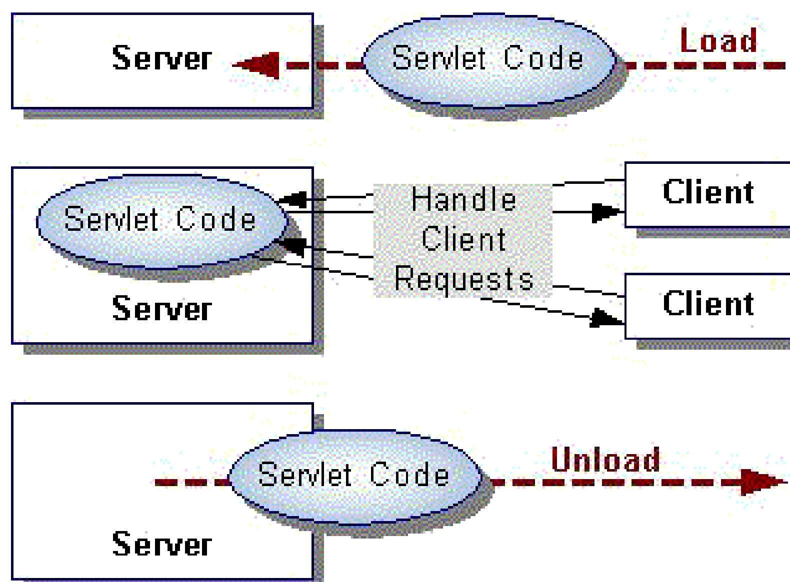


FIGURA 3.8 – Ciclo de execução de um servidor *servlet* [DAV 98].

- possuem todas as vantagens da linguagem Java, incluindo facilidade de desenvolvimento e independência de plataforma;
- podem ter acesso a um grande número de **APIs** disponíveis para a plataforma Java.

Servlets possuem um ciclo de vida que define como esta será carregada e inicializada, como irá receber e responder por requisições e como será finalizada. Em código, o ciclo de vida de uma *Servlet* é definido pela interface *javax.servlet.Servlet* (disponível após instalação do **JSDK**), sendo que todas as *Servlets* devem implementar direta ou indiretamente essa interface que é executada na *Servlet Engine*.

A *Servlet Engine* instancia e carrega uma *Servlet*. A instanciação e carregamento podem acontecer quando a *Servlet Engine* inicia, quando é necessário uma *Servlet* para responder por uma requisição ou a qualquer momento dentro disso. Ela pode carregar uma *Servlet* de um sistema de arquivos local ou de sistema de arquivos remoto. A figura 3.9 apresenta os estágios de execução de uma *servlet*.

3.3.3.1 Arquitetura dos *Servlets*

Quando um *servlet* aceita a chamada de um cliente, ele recebe dois objetos como parâmetros: o *ServletRequest* e o *ServletResponse*. O primeiro encapsula as mensagens do cliente para o servidor e o segundo encapsula as mensagens do servidor para o cliente. Existem subclasses das classes já citadas que permitem a manipulação de dados mais específicos para determinados protocolos, como é o caso das classes *HttpServletRequest* e *HttpServletResponse*.

A interface *ServletRequest* permite que o *servlet* acesse informações como os nomes dos parâmetros enviados pelo cliente, o protocolo que está sendo usado e o nome do *host* remoto que fez a requisição. Ainda é provido o *ServletInputStream*, através do qual o *servlet* pode obter dados do cliente, utilizando os métodos POST e PUT do protocolo HTTP.

A interface *ServletResponse* dá ao *servlet* a capacidade de responder ao cliente. Ela permite que o *servlet* especifique o tipo de dado que será enviado, provendo um objeto de saída *ServletOutputStream* e um outro objeto, *Writer*, pelos quais os dados são respondidos.

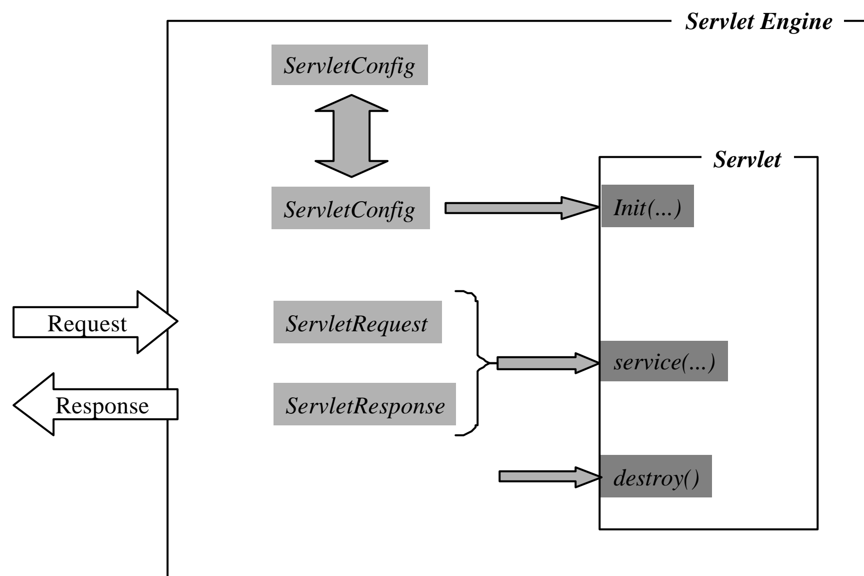


FIGURA 3.9 – Estágios de execução de um *servlet* [DAV 98].

3.3.4 RMI – *Remote Method Invocation*

Segundo [BUR 97] RMI (*Remote Method Invocation*) habilita a chamada remota de métodos. Também, possibilita que o método de um objeto, em uma máquina virtual, chame um método de outro objeto, em outra máquina virtual. Isto é feito com a mesma sintaxe e facilidade de uma invocação de objetos locais. Com RMI, um programa cliente pode invocar um método em um objeto remoto, da mesma forma que faz com um método de um objeto local. Todos os detalhes de conexão de rede estão ocultos, permitindo, assim, que o modelo de objetos tenha sua interface pública mantida através da rede, sem necessitar expor detalhes irrelevantes como conexões, portas ou endereços [ROC 98].

De acordo com [REI 99], RMI trabalha basicamente com um programa servidor, um programa cliente e um programa de interface, responsável pela interligação entre o cliente e o servidor. O computador servidor é utilizado para definir o corpo de cada um dos métodos que poderão ser executados remotamente. Um passo importante para que este método seja executado com eficiência é propiciar o registro de um ou mais objetos com o *rmiregistry* que deverá estar presente no servidor. Com isso, o cliente poderá acessar os métodos dos objetos das aplicações ativas no servidor. Neste processo é indispensável à definição de uma interface que especifica quais os métodos que poderão ser executados remotamente pelo cliente no servidor.

3.3.4.1 Arquitetura RMI

Quando se utiliza a RMI, no cliente é disponibilizada uma classe chamada *Stub*, e no servidor, uma classe chamada *Skeleton.* Essas duas classes são geradas por um programa fornecido como parte do JDK (*Java Developers Kit*). A classe *Stub* aparece para o cliente como se fosse um objeto real, permitindo a chamada a cada um dos métodos. Quando um método é chamado via *Stub*, os parâmetros são passados ao servidor via serialização e chegam até a classe *Skeleton*, que tem a finalidade de pegar os valores passados e efetuar a chamada de função do objeto real. O retorno do objeto real é transmitido do *Skeleton* para o *Stub*, e o cliente não faz a menor idéia de que o objeto está em outro local [RMI 97] e [RMI 2000].

Como primeira etapa para a criação de uma classe acessada via RMI, precisa-se de interfaces para o objeto remoto. Esta classe deve possuir a interface pública e todos os seus métodos têm de lançar uma *RemoteException().* Além disso, a interface da classe definida deve ser estendida da classe *Remote()*, parte da arquitetura RMI. Criadas e implementadas as Interfaces, deve-se compilar os códigos e gerar os arquivos *.class*. Estes serão utilizados pelo utilitário RMIC para a criação de classes *Skeleton* e *Stub*. O compilador utiliza-se dos arquivos compilados e, por isso, eles são necessários.

Cada objeto criado anteriormente a ser partilhado deve passar por um processo especial no servidor, consistindo de um registro, que dará um nome a cada objeto. O programa que gerencia todo o registro e acesso de nomes no servidor é o *rmiregistry*. Ele deve estar em execução (modo *stand by*) quando o servidor tentar registrar o objeto, bem como quando o objeto for acessado. Se algum objeto for solicitado ao servidor ele cria, executa, registra e termina a execução deste objeto instanciado através de uma chamada feita pelo cliente. Este, por sua vez, deve seguir a convenção de uma URL (*Uniform Resource Locator*), onde o protocolo RMI deve fornecer um servidor e o nome do método a ser executado remotamente pelo cliente no servidor.

O sistema RMI consiste de três camadas (figura 3.10):

- a camada de *stub/skeleton – stubs* do lado cliente (*proxies*) e *skeletons* do lado servidor;
- a cama de referência remota – comportamento de referência remota (como invocação para um único objeto ou para um objeto reproduzido);
- a camada de transporte – configura a gerencia, conexão e localização do objeto remoto.

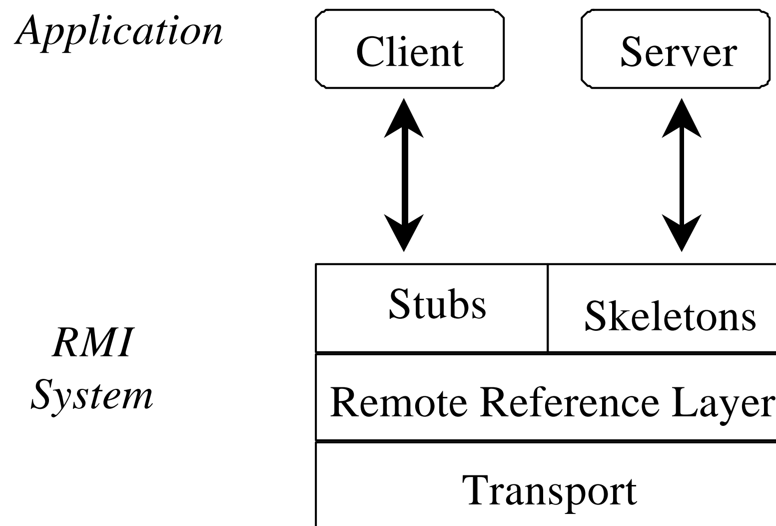


FIGURA 3.10 – Camadas do sistema RMI [RMI 97].

3.3.5 *Stream* de I/O para arquivos

Uma *stream* de I/O gera o caminho por meio do qual seus programas podem enviar uma seqüência de *bytes* de uma fonte até um destino. Uma *stream* de entrada é uma fonte (ou produtor) de *bytes* e uma *stream* de saída é o destino (ou consumidor) deles [PAP 91].

Todas as funções de I/O *stream* tratam os arquivos de dados ou os itens de dados como uma corrente ou um fluxo (*stream*) de caracteres individuais. Se for escolhida uma função de *stream* apropriada, a aplicação poderá processar dados em qualquer tamanho ou formato, desde simples caracteres até estruturas de dados grandes e complicadas. Tecnicamente falando, quando um programa usa uma função *stream* para abrir um arquivo para I/O, o arquivo que foi aberto é associado com uma estrutura do tipo ***FILE*** que contém informações básicas sobre o arquivo. Uma vez aberta a *stream*, é retornado um ponteiro para a estrutura ***FILE***. Este ponteiro ***FILE*** – algumas vezes chamado de *ponteiro stream* ou *stream* – é usado para fazer referência ao arquivo para todas as entradas/saídas subseqüentes.

Todas as funções de I/O *stream* fornecem entrada e saída bufferizada, formatada ou não formatada. Uma *stream* bufferizada proporciona uma localização de armazenamento intermediária para todas as informações que são provenientes de *stream* e toda saída que está sendo enviada a *stream*. I/O em disco é uma operação que toma tempo, mas a bufferização da

stream agilizará sua aplicação. Ao invés de introduzir os dados *stream*, um caractere ou uma estrutura a cada vez, as funções I/O *stream* acessam os dados em um bloco de cada vez. À medida que a aplicação necessita processar a entrada, ela simplesmente acessa o *buffer*, o que é um processo muito mais rápido. Quando o *buffer* estiver vazio, será acessado um novo bloco de disco. O inverso também é verdadeiro para saída *stream*. Ao invés de colocar fisicamente na saída todos os dados, à medida que é executada a instrução de saída, as funções de I/O *stream* colocam todos os dados de saída no *buffer*. Quando o *buffer* estiver cheio, os dados serão escritos no disco.

Dependendo da linguagem de alto nível que estiver sendo utilizada, pode ocorrer um problema de I/O bufferizado. Por exemplo, se o programa executar várias instruções de saída que não preencham o *buffer* de saída, condição necessária para que ele fosse descarregado no disco, aquelas informações serão perdidas quando terminar o processamento de programa. A solução geralmente envolve a chamada de uma função apropriada para "limpar" o *buffer*. É claro que uma aplicação bem escrita não deverá ficar dependendo destes recursos automáticos, mas deverá sempre detalhar explicitamente cada ação que o programa deve tomar. Mais uma observação: se a aplicação terminar de maneira anormal quando se estiver usando *stream* I/O, os *buffers* de saída podem não ser esgotados (limpos), resultando em perda de dados.

O último tipo de entrada e saída é chamado "*low-level* I/O" – I/O de baixo nível. Nenhuma das funções de I/O de baixo nível executa bufferização e formatação. Ao invés disso, elas chamam diretamente os recursos de I/O do sistema operacional. Estas rotinas permitem acessar arquivos e dispositivos periféricos em um nível mais básico do que as funções *stream*. Os arquivos abertos desta maneira retornam um manipulador de arquivo, um valor inteiro que é utilizado para fazer referência ao arquivo em operações subseqüentes. Em geral, é uma prática ruim de programação misturar funções de I/O *stream* com rotinas de baixo nível. Como as funções *stream* são bufferizadas e as funções de baixo nível não, a tentativa de acessar o mesmo arquivo ou dispositivo por dois métodos diferentes leva à confusão e à eventual perda de dados nos *buffers*.

3.3.5.1 *Stream* em Java

A biblioteca Java para I/O (**java.io.***[2]) oferece numerosas classes de *stream*. Mas todas as classes de *stream* de entrada são subclasses da classe abstrata *InputStream*, e todas as classes *stream* de saída são subclasses da classe abstrata *OutputStream*. A figura 3.11 ilustra as classes utilizadas para leitura e gravação de arquivos através de *stream* em Java [CHA 98].

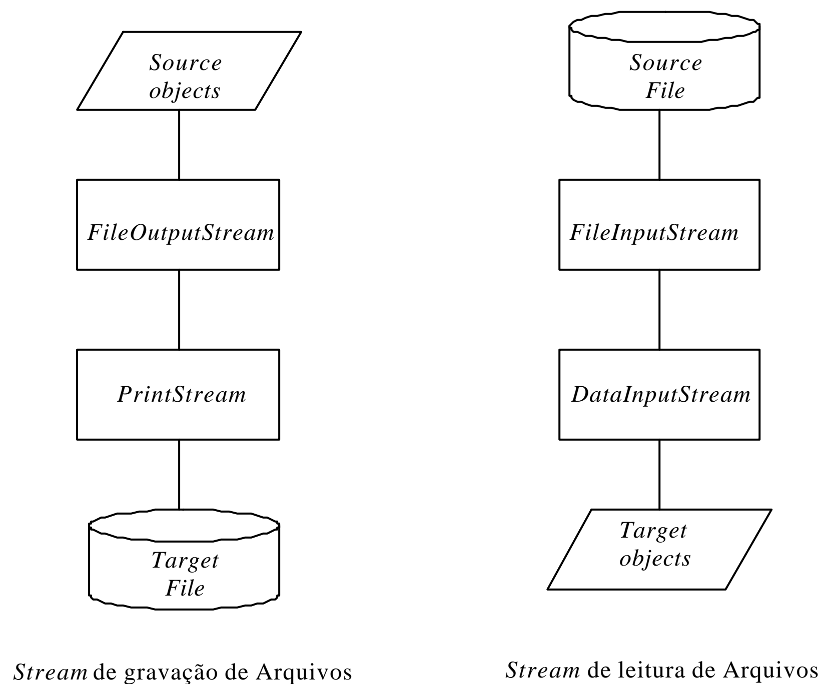


FIGURA 3.11 – Esquema de classes para I/O *stream* em arquivos [CHA 98].

O recurso de serialização permite que um objeto seja transformado em um *stream* de *bytes*, que podem ser armazenados em um arquivo e enviado via rede. Através desses *bytes* torna-se possível recriar a classe serializada, desde que esta implemente a interface *Serializable* ou a *Externalizable*. Somente objetos que implementem essas interfaces podem ser "desintegrados" em *bytes*, ou "integrados" a partir dos *bytes*. A interface *Externalizable*, por exemplo, deve

---

[2] Consulte www.java.sun.com/products/ para maiores informações sobre manipulação de arquivos através de *stream* em Java.

implementar os métodos *writeExternal* e *readExternal*. As variáveis serão salvas ou lidas por esses métodos. Essas duas chamadas recebem uma classe do tipo *ObjectOutput* e *ObjectInput*, respectivamente. O principal método da classe *ObjectOutput* é *writeObject* e o da classe *ObjectInput* é o *readObject*. Esses métodos serão os responsáveis pela transformação da classe em *bytes* e vice-versa. Existem classes concretas para essas interfaces, que são *ObjectOutputStream* e *ObjectInputStream.* Este recurso, serialização, pode ser utilizado em aplicações visuais, por exemplo, para permitir que os usuários salvem seus trabalhos em um arquivo. Neste caso, antes de escrever um objeto em uma *stream*, os objetos deverão ser serializados. Deve-se considerar a serialização de objetos semelhante à divisão de um objeto em pequenos pedaços para poder escrevê-los em uma *stream* de *bytes*. O termo para este processo é *serialização de objetos*.

4 Uma proposta de arquitetura para Ambientes Distribuídos de Edição Colaborativa

O compartilhamento de documentos entre usuários através de sistemas com características distribuídas permite a colaboração durante o processo de geração documentos. A intenção é disponibilizar uma ferramenta que auxilie no processo de geração distribuída de documentos através de editores diagramáticos. Neste capítulo serão apresentadas as principais características de colaboração, o modelo de colaboração adotado [SOU 97], a descrição da proposta de arquitetura de um *framework* para Editores distribuídos de Manipulação Colaborativa de Documentos Diagramáticos, o FreDoc, o os requisitos de tais sistemas.

4.1 Ambientes colaborativos

Segundo Otsuka [OTS 97] um ambiente colaborativo deve reunir funcionalidades que apóiem as seguintes atividades principais: comunicação, negociação, percepção, coordenação, compartilhamento, construção colaborativa de conhecimentos, representação de conhecimentos, e avaliação colaborativa. A seguir são apresentadas considerações sobre cada uma destas atividades colaborativas.

4.1.1 Principais atividades de um ambiente colaborativo

4.1.1.1 Comunicação

A comunicação é a mais importante característica das atividades em grupo, sem ela não há colaboração. É durante a comunicação que ocorre troca de idéias, discussões e conflitos entre os pares. Sendo à base das interações sociais, a comunicação é responsável, segundo os sócio-construtivistas, pela catalização do processo de desenvolvimento cognitivo individual. Segundo a corrente sócio-cultural, é através da comunicação que ocorre o desenvolvimento a nível interpsicológico, a partir do qual conceitos são internalizados e ativamente transformados pelo indivíduo, através da reflexão, constituindo assim o desenvolvimento a nível intrapsicológico. Já, de acordo com a corrente da cognição compartilhada, a comunicação e todo o contexto social que envolve os seus participantes, permite a construção e manutenção de conceitos compartilhados, os quais são considerados produtos do grupo todo.

A comunicação entre os participantes de um grupo pode ser apoiada por ferramentas de comunicação **síncronas** e **assíncronas**, através das redes de computadores. A primeira ocorre entre os usuários que estão ativos no sistema num mesmo momento, podendo ser realizada através de trocas de mensagens textuais (como no *talk e chat*) ou através de áudio e vídeo (videoconferências) já a segunda não exige esta coordenação temporal, os participantes podem realizar suas contribuições no momento em que julgarem mais apropriado, o que soluciona um dos grandes problemas do trabalho em grupo que é a incompatibilidade do horário de trabalho entre os participantes de um grupo. A comunicação assíncrona pode ocorrer através de mensagens textuais (como no correio eletrônico, *news e* documentos/hiperdocumentos), ou através de áudio e vídeo gravados.

Estes dois conceitos implicam diretamente no modo como se dá a consciência do grupo. Neste aspecto, identifica-se a **colaboração transparente** e a **colaboração consciente**. Na colaboração transparente as alterações realizadas por determinado usuário só são percebidas pelos demais quando alguém acessa estas informações, enquanto que na colaboração consciente há um mecanismo para notificar todos os envolvidos no projeto de que alguma alteração ocorreu.

Quanto ao local a comunicação pode ser no **mesmo** ou em **diferentes** lugares. As dimensões espaço-temporais são as principais em qualquer análise de atividade colaborativa auxiliada por computador. A liberação de restrições impostas por essas dimensões deve-se ao avanço da tecnologia de redes de computadores. Quer trabalhando em um mesmo local quer em locais distantes as redes são parte essencial da atividade de CSCW. Para Barros [BAR 94], combinando as dimensões tempo e espaço tem-se diferentes modalidades de interação entre os participantes de um grupo, conforme mostra a tabela 4.1.

TABELA 4.1 – Taxionomia Espaço Temporal [UNA 91].

Taxionomia espaço-temporal	Mesmo tempo	Tempo diferente
Mesmo lugar	Interação face a face	Interação assíncrona
Lugar diferente	Interação distribuída síncrona	Interação distribuída síncrona

Quanto ao modo a comunicação pode ser **comunicação direta** ou **interação indireta**. A comunicação direta (ou explícita) se dá através da transmissão de texto, gestos, vídeo e/ou áudio entre os participantes (é o que se chama normalmente de comunicação). A interação indireta (ou implícita) se dá através do objeto de trabalho (por exemplo, um texto ou imagem compartilhados).

Quanto à forma a comunicação pode ser **livre** ou **estruturada**, na forma livre a comunicação se dá através de conotações simples - frases, textos, desenhos, etc; lançadas de forma livre e sem estruturação. Porém dependendo do objetivo e da duração dessa comunicação, pode ser gerada uma grande quantidade de informação e que será alvo de referências. Nesse caso é desejável dispor-se de mecanismos que auxiliem a organização da informação para que com isto se consiga um melhor entendimento e não seja desestruturado o seu conteúdo principal [BAR 94]. Portanto para uma melhor organização das informações deve-se optar por modelos de comunicação estruturada onde as mensagens seguem um formato, um padrão ou hierarquia predefinidos.

A forma de comunicação livre permite a realização de uma **comunicação informal**. Hoje em dia, os pesquisadores reconhecem a importância deste tipo de comunicação em um ambiente de trabalho. Este tipo de comunicação "acidental" ocorre quando, por exemplo, dois colegas se encontram no corredor e discutem um tópico de interesse comum, ou quando um grupo se reúne informalmente para o *coffee break* e conversa sobre algum assunto relacionado ao trabalho. Ao contrário da comunicação informal, a **comunicação formal** segue um modelo de conversação pré-estabelecido, que pode ser representado por alguma ferramenta formal. No entanto, é sabido que nem os padrões de comunicação e nem a estrutura dos grupos são estáveis no decorrer de um trabalho colaborativo, o que torna estes modelos de conversação bastante limitados e de pouco "realismo social". Por esta razão, tem sido cada vez mais comum o

desenvolvimento de sistemas oferecendo oportunidades para a realização de encontros informais à distância, tanto para a comunicação entre indivíduos quanto para a comunicação entre grupos.

4.1.1.2 Negociação

A negociação é uma das principais características do trabalho em grupo e deve ser apoiada efetivamente por um ambiente de apoio a atividades colaborativas. As ferramentas de comunicação permitem que os participantes se comuniquem, mas não são suficientes quando é necessário que o grupo tome alguma decisão em conjunto. Portanto, para auxiliar o grupo na tomada de decisões, de forma que esta satisfaça a maioria, os sistemas de CSCW devem prover ferramentas (tais como mecanismos para a geração de idéias, identificação de propostas e votação) que permitam a resolução de conflitos, através da negociação de propostas entre os participantes [BAR 94] e [DIE 96]. Através destas ferramentas, cada integrante do grupo defende sua opinião e expõe seus argumentos que são lidos pelos colegas e contra-argumentados até que se chegue num consenso.

Em uma negociação estão envolvidos vários mecanismos cognitivos e afetivos - lógica, inferência, dedução, crença, dúvida, sutileza e envolvimento emocional com o assunto e com os participantes. O contexto de uma negociação - o tema, o objetivo, as questões, o “custo” das decisões, o tempo disponível, as pessoas envolvidas, o conhecimento comum, as hierarquias, os locais de reunião, as características culturais - são fatores que devem ser cuidadosamente observados para se fazer uma opção tecnológica de instrumento de suporte ao processo.

Quanto à forma uma negociação pode ser **livre** ou **estruturada,** no formato livre o computador é utilizado apenas para registrar e distribuir os argumentos usados na negociação, tais argumentos são expostos de maneira livre e sem nenhum formato preestabelecido. Porém em negociações mais complexas este modelo dificulta a percepção dos relacionamentos entre os argumentos e posicionamentos. Nestes casos a negociação estruturada é mais indicada, pois permite que sejam mais facilmente percebidos os inter-relacionamentos entre as falas da negociação. Nestes modelos os argumentos e contra-argumentos são expostos, inseridos, alterados e excluídos de forma estruturada seguindo um formato, um padrão, uma hierarquia ou seqüência de procedimentos predefinidos.

Quanto à coordenação uma negociação poderá ser **livre** ou **orientada**, no formato livre todos têm os mesmos direitos, prioridades e "papéis", já numa coordenação orientada são

definidos "papéis" com direitos e prioridades diferentes para cada participante da negociação, tal diferenciação é necessária em negociações mais complexas, com um grande número de participantes, a fim de se organizar e conduzir melhor a negociação.

4.1.1.3 Coordenação

A coordenação das atividades do grupo é fundamental para que os objetivos deste sejam alcançados de forma organizada, produtiva e harmoniosa. Segundo Dietrich [DIE 96], a necessidade de coordenação existe devido às interdependências de atividades, ou seja, quando a tarefa de um participante depende do que o outro participante está fazendo ou vai fazer, estes precisam coordenar e sincronizar suas atividades.

Segundo Dillengourg [DIL 94], a colaboração constitui uma "*atividade coordenada e sincronizada que é o resultado de uma tentativa contínua de se construir e manter um conceito compartilhado de um problema*".

A coordenação do grupo envolve o planejamento das atividades, a distribuição de tarefas e acompanhamento da execução destas. Na fase de planejamento é efetuada a divisão das tarefas que precisam ser realizadas para que o grupo alcance o seu objetivo comum, também são definidas metas e prazos a serem cumpridos. Após o planejamento é feita a distribuição das tarefas entre os participantes. A partir daí, tendo consciência não apenas dos seus compromissos, mas também dos compromissos de seus parceiros, os participantes podem cumprir suas tarefas de forma mais organizada, sendo possível também um melhor acompanhamento do cumprimento destas.

Muitas das atividades de coordenação podem ser feitas pelo próprio sistema como, por exemplo, registro de tarefas realizadas, envio de avisos sobre atividades atrasadas, dentre outros, liberando assim os coordenadores para atividades mais complexas.

4.1.1.4 Percepção

Nas atividades em grupo é fundamental que cada participante tenha percepção das ações dos demais participantes. A percepção fornece um contexto para as atividades individuais, contribuindo para uma maior sinergia do grupo[DIE 96].

Segundo Dietrich [DIE 96] existem duas formas principais de prover a percepção em *groupwares*: a explicitamente gerada e a passivamente colecionada e distribuída. Na forma explicitamente gerada, as ações são armazenadas pelo sistema e geralmente necessitam ser explicitamente informadas pelos participantes. As ações dos participantes são distribuídas a todos em formato de boletins informativos ou relatórios em horários pré-determinados ou quando solicitadas. Na forma passivamente colecionada e distribuída as ações são distribuídas em tempo real, à medida que vão ocorrendo.

4.1.1.5 Compartilhamento

As atividades de aprendizagem e trabalho colaborativo envolvem o compartilhamento de objetivos, idéias, descobertas, objetos e produtos destas atividades. Portanto, para o desenvolvimento de uma colaboração efetiva é necessária a criação de uma "memória organizacional do grupo" [BAR 94], acessível a todos os participantes. Nesta memória devem ser armazenados padrões, orientações, projetos, descobertas e resultados do trabalho que está sendo desenvolvido pelo grupo. Também devem ser registrados resultados de reuniões, decisões, planos de ações, e outras informações que orientem o desenvolvimento do trabalho do grupo, e auxiliem na aprendizagem colaborativa.

Quanto à simultaneidade o compartilhamento pode ser ao mesmo tempo ou em tempos diferentes.

4.1.2 Co-autoria Colaborativa de documentos

Nas seções anteriores foram feitas considerações sobre ambientes colaborativos, nesta seção serão feitas considerações mais específicas a ambientes de co-autoria colaborativa de documentos que são o foco desta pesquisa.

Segundo Grudin [GRU 94], o processo de *co-autoria* é definido como sendo a elaboração conjunta de documentos, onde membros de um grupo expõem suas idéias e a partir delas chegam a um consenso que se reflete na versão final do documento.

Os principais requisitos de sistemas de autoria colaborativa relativos ao processo de criação são: 1) composição de documentos através de mecanismos que permitam aos escritores expressarem suas idéias sobre documentos, constrangimentos e estratégias (tomada de nota, compartilhamento de idéias e discussão); 2) suporte a subgrupos: definição de usuários, acesso e privilégios; 3) ferramentas com mecanismo de controle de mudanças e distribuição de versão, além da habilidade para autenticar submissões e comentários (para dar crédito às contribuições e idéias); mecanismo de gerência e acesso a versões prévias de documentos; 4) o processo de comentário é uma grande atividade de comunicação dentro dos grupos de autoria e entre leitores e tais grupos; é preciso identificar comentaristas e revisores e também é necessário que comentários estejam associados com a porção apropriada do texto a que se referem; 5) registro de todas as atividades envolvidas no processo (*tracking*), identificando-se o responsável, data e hora da execução da atividade; 6) recuperação das informações sobre as atividades já desenvolvidas (*reporting*); isto é importante para que todos tenham consciência de como o projeto está se desenvolvendo; 7) função de treinamento para o grupo de trabalho; 8) ferramentas para reuniões locais e distribuídas; 9) ferramentas para gerência do projeto; (10) ferramentas de suporte disponíveis para os membros individuais; e 11) a colaboração pode ser síncrona ou assíncrona.

4.1.3 Implementação

As tecnologias para suporte à colaboração, podem ser classificadas em três categorias [SAP 2001]: 1) Ferramentas: tecnologias básicas que têm sido incrementadas para suportar colaboração; 2) Programas Integrados: tecnologias cuja função primária é suportar uma ou mais das características de colaboração; e 3) Ambientes: coleções de tecnologias e ferramentas sobre um único "ambiente" coordenado que suporta colaboração; estes sistemas são caracterizados, além da coleção de ferramentas por três aspectos: uma função de controle, normalmente oferecida através de algum tipo de base de dados, uma linguagem de "*scripting*" que permite especificar processos de trabalho a ser descrito, e uma interface gráfica que permite adicionar processos para acessar e compartilhar o armazenamento de informação.

Com relação à arquitetura utilizada na implementação de aplicações colaborativas, os sistemas podem ser ***hard-wired***, **centralizados**, **replicados** ou **híbridos** [SAP 2001]. Na configuração ***hard-wired***, os componentes da rede são construídos com o propósito de realizar funções específicas do sistema. Um exemplo seria um espaço de trabalho compartilhado

totalmente baseado em vídeo; as câmeras, monitores e projetores seriam interconectados para a geração, transmissão e projeção das imagens dos participantes e de seus trabalhos. A configuração **centralizada** segue o modelo cliente-servidor, onde todos os participantes só podem se comunicar com o servidor central, que realiza o processamento necessário e retransmite as mensagens. Dentre as vantagens da configuração centralizada estão sua relativa simplicidade de implementação, facilidade para garantir consistência no estado da aplicação (existe apenas uma cópia executando no servidor), além de permitir clientes mais simples e evitar os complexos algoritmos distribuídos. A configuração **replicada** ou distribuída executa uma cópia da aplicação em cada máquina. As vantagens desta configuração incluem a geração de menos tráfego na rede, já que as mensagens não precisam passar pelo servidor, e uma maior escalabilidade do sistema (o servidor se sobrecarregaria acima de um determinado número de clientes). Tentando unir as vantagens destas duas últimas configurações, surgiu a configuração **híbrida**. Nesta configuração, cada cliente executa uma cópia (completa ou parcial) da aplicação e há também a presença de um servidor central, responsável por tarefas como a garantia de consistência entre os clientes, sincronização e tratamento de colisão de eventos (por exemplo, dois usuários querendo editar a mesma parte de um documento).

4.1.4 Flexibilidade

Um ponto muito importante em ambientes colaborativos e característica essencial em ambientes de sucesso é a flexibilidade. Segundo [SCH 92], a flexibilidade de uma aplicação colaborativa pode ser analisada do ponto de vista **técnico**, **funcional**, do **campo de aplicação** e também do ponto de vista **social**. Flexibilidade **técnica** está relacionada à portabilidade e capacidade de adaptação do sistema nos diversos ambientes computacionais. Uma aplicação é considerada flexível tecnicamente se permitir que o trabalho colaborativo seja realizado em diferentes plataformas de hardware, sistemas operacionais, interfaces gráficas, com vários formatos de áudio e vídeo e diferentes dispositivos de entrada e saída. Flexibilidade **funcional** está relacionada à classificação espaço-temporal apresentada na seção 4.1.1.1. Um sistema que se enquadre nas várias categorias daquela taxionomia e permita o "chaveamento" entre os vários modos de interação e localização geográfica dos usuários é mais flexível no aspecto funcional. Flexibilidade do ponto de vista do **campo de aplicação** está relacionada à outra taxionomia; um sistema que suporte mais tipos de aplicações (de comunicação, de espaço de trabalho compartilhado, etc.) é mais flexível sob este aspecto.

A flexibilidade do ponto de vista **social** é bem mais complexa, tendo sido tema de vários trabalhos que tentam alertar os projetistas de sistemas colaborativos sobre as características dinâmicas e pouco previsíveis do trabalho em grupo [SCH 92]. A primeira dificuldade na modelagem do trabalho em grupo diz respeito às ações "localizadas". Cada participante, em certas circunstâncias, poderá se encontrar em situação única, desconhecida dos demais participantes e terá de lidar com ela individualmente (por exemplo, em caso de documentos perdidos, dados incompletos, quebra de equipamento, etc.). Para lidar com estas contingências locais, o participante muitas vezes se vê obrigado a violar o critério global, já que nenhum critério se aplica a todas as contingências [SCH 92]. Nestas situações, cada participante lidará com o problema segundo suas próprias heurísticas, que podem ser diferentes para cada um. Outro aspecto ainda mais importante diz respeito à consideração das diferenças de interesse, poder e *status* dentro de um grupo. É muito comum encontrar na literatura teorias e sistemas colaborativos que adotam uma visão utópica do trabalho em grupo, considerando apenas os aspectos positivos da colaboração e ignorando conflitos, competições, coerções, etc. A flexibilidade do ponto de vista social exige que a aplicação leve em consideração estes fatores.

Em resumo, para serem flexíveis, aplicações colaborativas não devem impor padrões de trabalho (ou comunicação) pré-estabelecidos. Na verdade, elas devem prover facilidades que permitam aos usuários interpretar e explorar estes modelos formais, mas deverá sempre caber ao usuário a decisão de usá-los, modificá-los ou rejeitá-los.

4.2 Sistemas Colaborativos

Segundo [FAR 98], um sistema colaborativo é composto por múltiplos usuários envolvidos em uma atividade compartilhada, normalmente em locais remotos. Dentre as aplicações distribuídas, sistemas colaborativos são diferenciados pelo fato de que os usuários do sistema estão sempre trabalhando com relação a um objetivo comum e têm necessidade de interagirem: dividindo informações; trocando solicitações e avaliando sua situação com outros usuários. Considera-se sistema colaborativo como um sistema que também possui um certo nível de concorrência entre usuários que buscam interação com o sistema. Assim, uma sessão *chat* é colaborativa, porque todos os usuários envolvidos precisam coordenar-se, para assegurar que os *chatters* não percam o comentário de ninguém. Um cliente de *e-mail* não tem cuidado sobre o estado de nenhum outro cliente, pois não precisa coordenar-se com ninguém para atingir seu

objetivo. Alguns dos principais elementos de um sistema colaborativo são: 1) interfaces com usuários; 2) servidores; 3) repositório de dados; e 4) transações entre usuários, servidores e repositório de dados, ilustrados na figura 4.1.

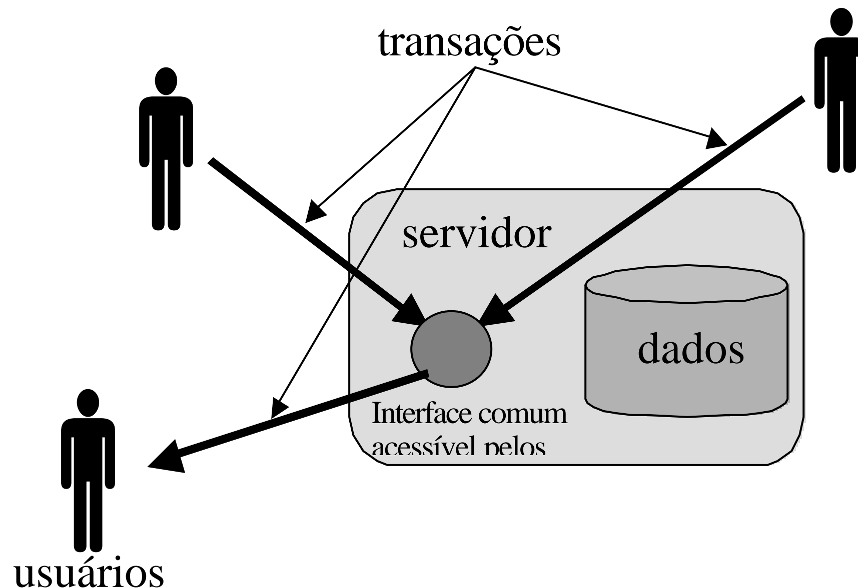


FIGURA 4.1 – Estrutura de um sistema colaborativo [FAR 98].

Interfaces com usuários, servidores, repositórios de dados e transações são elementos que geralmente compõe um sistema distribuído. As transações entre usuários e o compartilhamento dos objetos entre estes tornam-no um sistema colaborativo.

4.2.1 Necessidade de Comunicação

Um sistema colaborativo tem múltiplos usuários remotos colaborando dinamicamente devendo ser flexíveis em suas habilidades de direcionamento transações. Dependendo da aplicação, uma comunicação pode precisar suportar: 1) mensagens ponto-a-ponto entre usuários; 2) envio de mensagens "*broadcast*" para uma comunidade de usuários; ou 3) envio de mensagens "*narrowcast*" para usuários participantes de grupos específicos. Um servidor interativo de *chat* que suporta salas de *chat* é um bom exemplo disto. Normalmente é dado um *broadcast* para um grupo completo. Mas, se for uma sala de *chat* particular, a mensagem poderá ser enviada somente

para seus participantes. Em alguns casos, o sistema deverá manter comunicação privada "*unicast*", enviando mensagens entre dois usuários do sistema.

4.2.2 Mantendo Identificadores de Usuários

Segundo [FAR 98], deverão existir mecanismos para identificação individual de usuários colaboradores para que mensagens possam ser endereçadas e enviadas, trabalhos possam ser designados, etc. Se o acesso para o sistema ou para certos recursos associados com o sistema precisarem ser restritos, será necessário o uso de uma identificação, para o usuário ser melhor autenticado. Dependendo da aplicação, poderão existir dados ou outros recursos associados a cada usuário. Esta informação deverá ser mantida um identificador e, em alguns casos, o acesso sobre os recursos do sistema deverá ser controlado.

Como exemplo pode-se citar a aplicação de quadro compartilhado "*whiteboard*" [FAR 98]. Um quadro compartilhado é um espaço virtual de desenho no qual múltiplos usuários remotos podem normalmente ver e escrever, compartilhando informações, idéias, etc. Um exemplo real equivalente seria um grupo de pessoas reunidas em uma sala, trabalhando ao redor de um quadro. Certamente, para que um indivíduo que esteja usando o quadro entenda quem escreveu o que, o sistema terá que conservar algum tipo de identificador de cada participante. A contribuição deve ser mostrada com sua identificação tornando-se útil para que cada indivíduo possa corrigir, modificar ou apenas apagar suas contribuições.

No EDI (Editor de Diagramas para a Internet) [POM 99] especificou-se que a identificação de usuários ocorreria através de cores, associando as anotações inseridas em um modelo compartilhado ao seu autor.

4.2.3 Informação de Estados Compartilhados

Em sistemas colaborativos alguns dados e recursos são compartilhados entre vários participantes. Um esforço colaborativo entre múltiplos usuários é normalmente expressado tendo como base à quantidade de dados que serão compartilhados entre usuários de um sistema [FAR 98]. Por exemplo, no *whiteboard* o conteúdo atual é compartilhado entre os usuários. Manter a integridade da informação quando múltiplos usuários estiverem acessando e modificando o estado compartilhado desta, se torna uma questão importante. Se dois ou mais agentes tentarem alterar o mesmo "pedaço" de informação, então se deve ter uma maneira de fundir as múltiplas requisições e informar aos agentes afetados como foram feitas as transações.

4.2.4 Desempenho

Alguns sistemas colaborativos têm que fazer um *trade-off* entre manter a consistência dos estados compartilhados através de todos os usuários e maximizar desempenho. Existem situações, como no *whiteboard*, onde é importante que todos tenham a visão dos estados compartilhados. A forma mais simples de se fazer isso é ter um mediador central atuando como um distribuidor de eventos [FAR 98].

O problema é que o mediador central pode tornar-se lento à medida que o tamanho do sistema aumente. Se existirem muitos usuários para serem notificados, então o mediador pode ter problemas para manter o tráfego e os usuários do sistema perderão tempo esperando pelas atualizações. Um outro caminho seria usar um sistema ponto-a-ponto, onde cada usuário envia suas atualizações para todo o grupo. Apesar de cada atualização existir em um "*broadcast*" assíncrono e independente, o mediador garante que cada usuário finalize sobre o mesmo estado após todas as atualizações terem sido efetuadas, especialmente se a ordem das atualizações for importante.

4.3 Modelo de Colaboração

Este modelo foi proposto por [SOU 97] baseado em experiências provenientes da escrita colaborativa, fornecendo apoio às tarefas de: *revisão* (onde existe um autor e vários comentadores) e co-autoria (onde existem múltiplo autores, cada um deles com direitos iguais sobre o diagrama). O modelo baseia-se na utilização de anotações em diagramas. As anotações

são uma forma de comunicar idéias ou opiniões sobre o conteúdo de um diagrama e podem ser feitas sobre o diagrama como um todo ou sobre elementos específicos do mesmo. Por exemplo, na metodologia OMT [RUM 91], os elementos que podem ser anotados são classes, relacionamentos, atributos de classes, conjunto de classes, etc.

Uma anotação apresenta um escopo de visibilidade que corresponde ao conjunto de usuários que podem visualizá-los. O escopo pode ser genérico (quando qualquer comentador puder visualizar uma anotação) ou específico (quando somente um subconjunto de comentadores puder visualizá-la). O modelo permite que o autor possa comentar seu próprio diagrama justificando suas decisões, o que determina que os comentadores não precisem inferir sobre decisões de projeto tomadas pelo autor [NEU 90].

A comunicação entre os co-autores e comentadores através de anotações pode ser feita de várias formas: apresentando opiniões, propondo alterações, levantando questões, etc. Segundo [SOU 97] o modelo define os tipos de anotações utilizadas para atender a estas formas de comunicação, apresentadas nas subseções a seguir.

4.3.1 Comentário

Corresponde a um trecho de texto que, normalmente, está associado a elementos ou conjunto de elementos do diagrama. O comentário pode ser geral (não estar ligado a nenhum componente) ou específico (quando esta ligado a um ou vários componentes), podendo ser inserido pelo colaborador ou pelo proprietário de um diagrama.

4.3.2 Inclusão

Corresponde a uma proposta de inclusão de novos objetos no diagrama original. Quem decide sobre a inclusão definitiva destes novos objetos no diagrama é o proprietário. Inicialmente tais inclusões ficam associadas às cópias enviadas aos usuários colaboradores sendo posteriormente analisadas e atualizadas pelo proprietário.

4.3.3 Exclusão

Corresponde a uma proposta de exclusão de objetos de um diagrama. O usuário colaborador poderá excluir diretamente apenas os objetos por ele editados. Os demais objetos que não foram por ele editados, só serão excluídos caso o proprietário valide tal ação sobre seu diagrama.

4.3.4 Substituição

Corresponde a uma sugestão de alteração no diagrama original. Ela pode ser feita sobre um elemento ou um conjunto de elementos do digrama. Deve-se, no entanto, permitir que substituições não estejam associadas a elementos do diagrama, uma vez que o modelo proposto deve fornecer apoio a todas as fases do processo de desenvolvimento.

4.3.5 Proposta

Corresponde a um conjunto de anotações relacionadas. Neste caso, quando uma proposta é visualizada, todas as anotações que a compõem também são. A implementação de uma pilha a partir de uma lista, por exemplo, pode ser feita utilizando-se herança ou agregação. No entanto, se for utilizada agregação [RUM 91], deve-se necessariamente incluir um atributo do tipo lista na classe pilha, ou seja, a modificação no relacionamento entre lista e pilha dever ser acompanhada da inclusão de um atributo em pilha. A proposta seria formada por uma substituição no relacionamento de herança por um relacionamento de agregação e uma substituição que inclui um novo atributo na pilha. Esta é uma situação comum no desenvolvimento de sistemas, onde a modelagem de um conceito implica em alterações em todo o diagrama.

O uso do modelo baseia-se em três estágios: *criação*, *discussão* e *incorporação*. No estágio de criação o autor efetua a construção do modelo de objetos. Para dar apoio a esta fase, a funcionalidade de um editor colaborativo é similar à funcionalidade de um editor comum de diagramas, logo deve-se permitir funções como: inclusão e exclusão de classes, relacionamentos e métodos, etc. No estágio de discussão os comentadores inserem as anotações no diagrama. Logo o editor deve fornecer apoio à manipulação de anotações. A inserção de anotações é feita

através da seleção de um conjunto de elementos do diagrama, definidos pela notação que o editor apóia. No estágio de incorporação o autor avalia as anotações que foram inseridas, podendo incorporá-las ou não ao editor como, por exemplo, mecanismos de visualização das anotações, mecanismos de indicação de sua existência, etc. A indicação da existência das anotações deve ser feita de modo não-intrusivo, utilizando-se para isso uma *coluna de anotações* [NEU 90]. Se um co-autor deseja visualizar uma anotação, basta selecionar a anotação a ser apresentada. Caso contrário, ele pode editar o diagrama normalmente.

Segundo [REE 92], as anotações nesta coluna são apresentadas através de *filtros* utilizados como um meio para restringir a apresentação. A visualização efetiva das anotações deve ser feita através de algum mecanismo que permita a apresentação das anotações no contexto original onde elas foram criadas [NEU 90]. Assim, quando uma substituição é aplicada sobre um diagrama deve-se retirar os elementos sobre os quais ela foi efetuada e em seu lugar apresentar os elementos que compõem a substituição. No entanto, o número de elementos que compõem a substituição pode ser maior que o número de elementos originais, ou ainda, eles podem apresentar uma disposição diferente da original. Uma solução para tais problemas é a utilização de algoritmos para redesenho automático de diagramas [SIM 89].

Após a definição inicial do modelo de objetos os estágios de criação e incorporação podem ser mesclados, de tal forma que o autor pode incorporar anotações ao diagrama e/ou inserir novos elementos.

4.4 Metodologia do projeto

O desenvolvimento de sistemas colaborativos, sendo uma área de desenvolvimento, requer ambientes de desenvolvimento extensíveis, com múltiplos modos de utilização e vários níveis de abstração. Essas características levam à construção de sistemas complexos devido, essencialmente a análise de [BOO 91] e que, segundo [WEG 96], não podem ser facilmente resolvidas por técnicas de decomposição como a análise estruturada.

Gamma et al. [GAM 95] notaram que bons projetistas freqüentemente utilizam soluções baseadas em experiências passadas. Um projetista experiente tende a aplicar recorrentemente soluções comprovadamente eficazes que acabam por gerar sistemas flexíveis e

extensíveis. Essas soluções são chamadas de *design patterns*, ou simplesmente padrões. As ferramentas mais populares de projeto utilizam algum tipo de decomposição, de forma que os componentes de um sistema possam ser analisados isoladamente. Padrões descrevem componentes e também relações entre componentes. Por essa razão, padrões são realmente considerados uma técnica arquitetônica e não uma técnica de divisão-e-conquista como a análise estruturada.

Brugali et al. [BRU 97] apontam que, com o auxílio de padrões, novas aplicações desenvolvidas a partir de um *framework* produzem novos componentes de suporte. Isso significa que, na sua concepção, um *framework* é predominantemente caixa-branca e, com o tempo, o desenvolvimento de componentes (subsistemas) gera soluções tipo caixa-preta que estendem sua funcionalidade.

Pensa-se então que, um sistema de edição colaborativa de documentos diagramáticos deve oferecer um ambiente rico em ferramentas que auxiliem no processo de especificação de componentes visuais gráficos a serem utilizados na edição de documentos, um bom controle de acesso aos recursos disponíveis, visando identificar os colaboradores durante um processo colaborativo, mecanismo de gerenciamento de informações que busquem a centralização de documentos em um repositório de dados e, principalmente, estar disponível para acesso através de ambientes distribuídos e heterogêneos, como a internet.

4.5 Requisitos da arquitetura do FreDoc

O *framework* FreDoc foi projetado a partir da identificação de requisitos de ferramentas como EDI [POM 99] e *HotDraw* [JOH 92], que são respectivamente, um ambiente baseado na *internet* para desenvolvimento colaborativo de *software* e um *framework* para criação de ambientes de edição colaborativa via *web*. Como resultado da análise destas ferramentas, verificou-se que um editor distribuído de manipulação colaborativa de documentos via *web* deve apresentar as seguintes características: 1) estar disponível para acesso via *web* (portabilidade); 2) definir características visuais proprietárias do usuário, visando facilitar a identificação dos colaboradores de um determinado documento; 3) armazenar documentos em uma linguagem de definição bastante genérica que possibilite a utilização destes por outras ferramentas; 4) armazenar documentos gerados pelo editor visando facilitar o processo de colaboração em

ambientes distribuídos; 5) possuir ferramentas para edição de diagramas, tanto textual como diagramático; 6) possuir ferramentas de controle de edição visual dos objetos gráficos a serem apresentados ao usuário; 7) possuir mecanismos auxiliares de comunicação (*e-mail*, *chat*, *voice*); e 8) aceitar formatos de outros editores.

Desta forma, pode-se dar ao usuário algumas facilidades, tais como: 1) manipulação de diferentes tipos de documentos; 2) colaboração; e 3) possibilidade dos usuários estarem distribuídos na *internet*.

As duas ferramentas citadas anteriormente, o EDI e o HotDraw contemplam quase todas as características enumeradas, sendo que o EDI implementa as características 1, 2, 4, 5, 6 e 7, enquanto que o HotDraw, por ser mais genérico, implementa apenas as características 1, 5 e 6, deixando a cargo de quem for utilizá-lo implementar ou não as demais. Já o FreDoc, por nestas ferramentas, herdou as propriedades já implementadas pelo EDI e pelo HotDraw, acrescido do item 3.

4.6 Arquitetura do FreDoc

FreDoc foi desenvolvido utilizando-se recursos descritos nos capítulos e seções anteriores. Ele é um *framework* que disponibiliza um ambiente distribuído para edição colaborativa de documentos diagramáticos. Para que isso seja possível, ele está implementado em camadas, utilizando-se de comunicação cliente-servidor através de *servlets*, com padrão de aplicação *thick web client* e suporte para edição de documentos diagramáticos. Essas características estão descritas a seguir.

O FreDoc foi construído baseado na arquitetura em camadas, tendo sido especificado três componentes fundamentais: 1) programas escritos em java e páginas HTML como componentes de apresentação; 2) *servlets* java como componentes de negócio; e 3) conexão JDBC:ODBC do java como componente de acesso a dados.

Utiliza-se de *servlets* java para comunicação entre o cliente e o servidor, possibilitando a execução remota de métodos, que em sua estrutura disponibilizam acesso a banco de dados.

Baseado no padrão *thick web client* de aplicações para *web*, que prevê a execução e controle de algumas transações pelo cliente e interfaces melhoradas, com mais recursos à sua disposição.

Prevê suporte para a edição de documentos diagramáticos, possibilitando o uso de componentes básicos de edição gráfica (linha, texto, polígono) através de um ambiente visual gráfico para edição colaborativa via *web*.

O protótipo do FreDoc, o FreDocUML, foi implementado em java, reutilizando classes de sua biblioteca de classes e também do *framework HotDraw for java* [KNO 97] para o suporte ao desenvolvimento de editores gráficos.

O *framework* FreDoc faz uso extensivo das facilidades da API padrão do JDK (*Java Development Kit*), na versão 1.3. Ele é constituído por quatro pacotes de classes que complementam a funcionalidade do JDK e, em especial, do AWT (*Abstract Window ToolKit*). Utilizou-se também a API do JSDK (*Java Servlet Development Kit*) utilizada para acesso remoto à banco de dados via *web* e o *framework HotDraw for Java*[3] [KNO 97] que possui um conjunto de classes para a criação de editores gráficos.

Considerando as questões de reuso e composição, adotou-se o paradigma de orientação a objetos e padrões bem difundidos [GAM 95], descritos no capítulo 3, utilizados, por exemplo, na construção de *frameworks* como o *HotDraw* [JOH 92], *HotDraw for Java* [KNO 97] e mais recentemente pelo EDI [POM 99]. Adotou-se também o padrão *Thick Web Client* de arquitetura para aplicações *web*, descritos na seção 3.2.2.

O *framework* está organizado de acordo com o padrão MVC [BUS 96]. Suas classes definem os relacionamentos que determinam a forma de composição de aplicações. As classes *Model*, *View* e *Controller* são utilizadas para a construção de *interfaces* em *Smalltalk*. Objetos desses tipos representam, respectivamente, a aplicação (*Model*), a *interface* (*View*), e o mapeamento das funções da *interface* para as funções da aplicação (*Controller*). Essa decomposição, além de promover a reutilização dos componentes, imprime maior flexibilidade à aplicação. Cada modelo se ocupa de notificar mudanças no seu estado às visões associadas que, por sua vez, têm a oportunidade de atualizar sua aparência em função do estado do modelo. Dessa forma, cada modelo pode ser representado por múltiplas visões sem que a modificação ou criação de novas visões implique modificações no modelo. O controle executa métodos do

[3] para maiores informações sobre os pacotes do *framework HotDraw for Java* deve-se consult ar http://www.ksccary.com.ksc/HotDraw.HotDraw

modelo em função de eventos da *interface*. As classes *ShapeTool*, *Shape*, *SimpleDrawingCanvas* fazem, respectivamente, os papéis de controle, visão e modelo, conforme figura 4.2.

O ponto principal do *framework* é a classe *SimpleDrawingCanvas* que contém um *canvas* para exibição e composição de diagramas, gerência de operações sobre objetos, controle das estruturas de dados de armazenamento, *scrolling* e controle sobre as transações locais de documentos. A classe *Shape* é a superclasse para representação de objetos gráficos que obriga todas as subclasses a implementar uma *interface* comum baseadas em formas geométricas básicas (linhas, retângulos, polígonos e até texto). Esta classe prevê apenas objetos gráficos com duas dimensões.

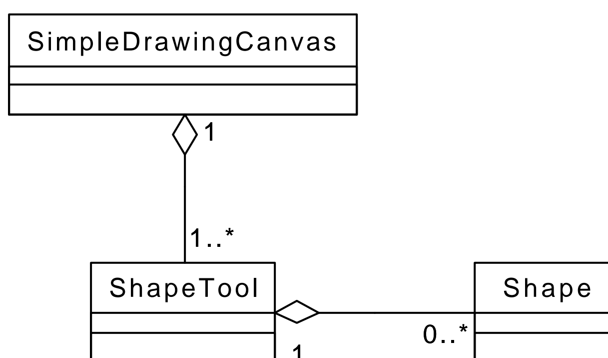


FIGURA 4.2 – Modelo MVC implementado para o FreDoc.

Operações sobre objetos gráficos que compõem um documento são implementadas como objetos da classe *ShapeTool*. A classe *ShapeTool* pode ser estendida para receber eventos do *canvas* e, portanto, pode ser utilizada para implementar funções de interação. Sua funcionalidade não está restrita a inserção de objetos na área de *canvas*, podendo ser utilizada também para definição de ferramentas para seleção e movimentação de objetos, por exemplo.

4.6.1 Pacotes da arquitetura proposta para o desenvolvimento de Editores Distribuídos de Geração Colaborativa de Documentos

Além das classes descritas na figura 4.2 para controle de visões de documentos editados ou em edição, também foram definidos alguns pacotes, figura 4.3, contendo classes para possibilitar o controle de outros recursos para o editor tais como: 1) estilo de edição; 2) classes base para formação de componentes gráficos do editor; 3) controle de acesso; e 4) estrutura para confecção de classes persistentes para armazenamento de objetos gráficos em edição.

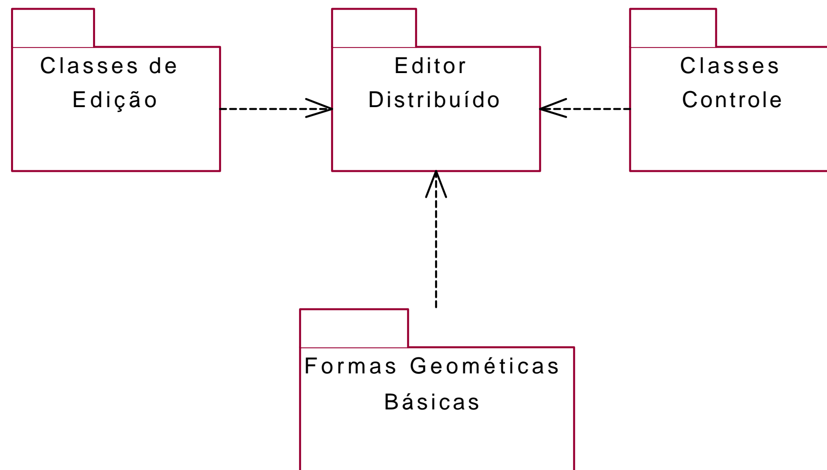


FIGURA 4.3 – Pacotes da arquitetura proposta para o desenvolvimento de Editores Distribuídos de Geração Colaborativa de Documentos.

As classes do pacote **Classes de Edição**, figura 4.4, são utilizadas como base para a definição do estilo de visualização do documento em edição para um determinado usuário.

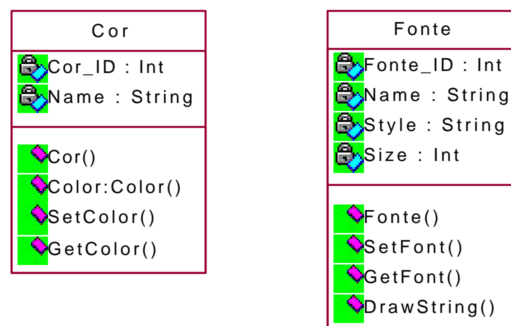


FIGURA 4.4 – Diagrama de Classes do Pacote Classes de Edição.

Já o pacote **Formas Geométricas Básicas**, figura 4.5, é utilizado para compor objetos gráficos a serem utilizados no processo de edição de documentos. O pacote **Classes de Controle de Acesso**, figura 4.6, será utilizado para autenticar e controlar o acesso de usuários a editores construídos através da arquitetura proposta. Por último, o pacote **Editor Distribuído**, figura 4.7, constituído de classes utilizadas no controle de documentos em edição, se preocupando com o autor, projeto relacionado e objetos gráficos utilizados durante sua confecção.

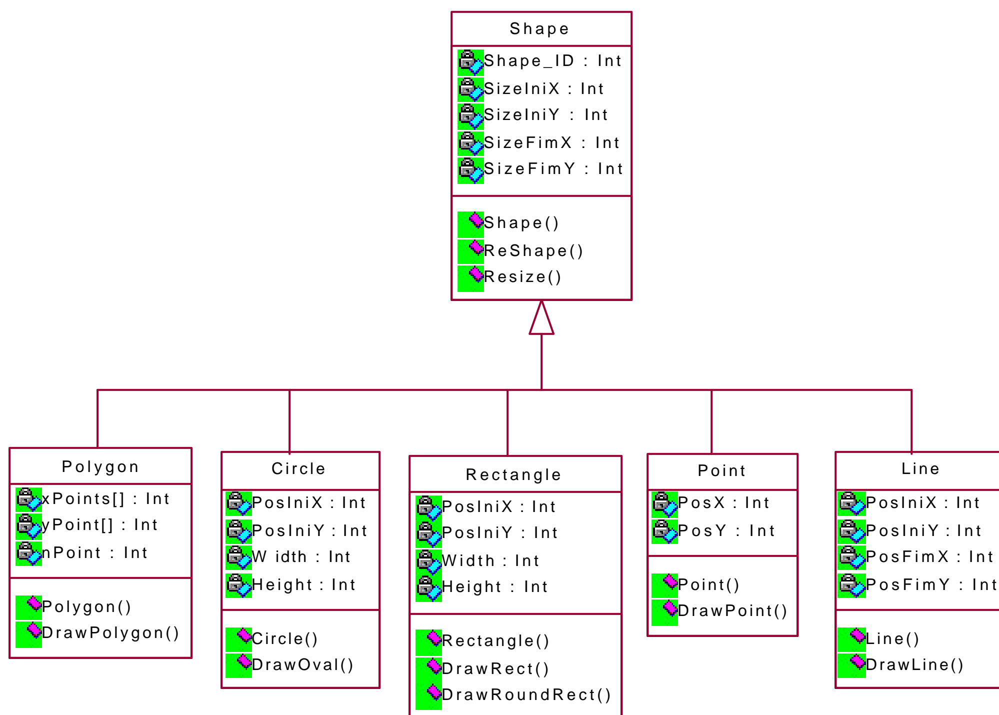


FIGURA 4.5 – Diagrama de Classes do Pacote Formas Geométricas Básicas.

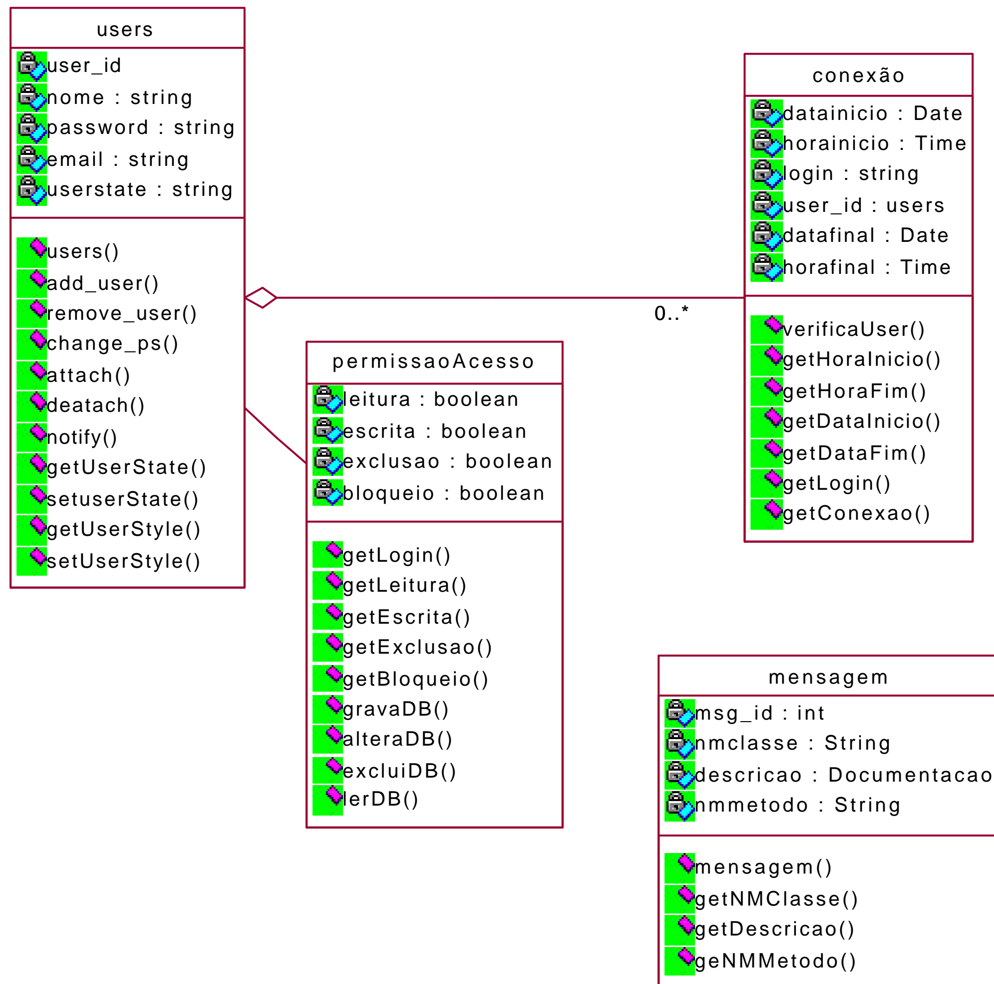


FIGURA 4.6 – Diagrama de Classes do Pacote Classes de Controle de Acesso.

4.6.2 Descrição das principais classes implementadas na arquitetura do FreDoc

Nesta seção será descrita a modelagem da proposta sugerida. Apresentar-se-á também o modelo de classes implementado e os recursos utilizados para especificação e desenvolvimento, da arquitetura ilustrada anteriormente pela figura 4.2.

A Proposta de arquitetura para ambientes distribuídos de edição colaborativa de documentos[4] foi implementada como um conjunto de pequenas ferramentas que agregadas formam um editor para criação de manutenção de documentos. A figura 4.8 apresenta as principais classes que compõem esta arquitetura.

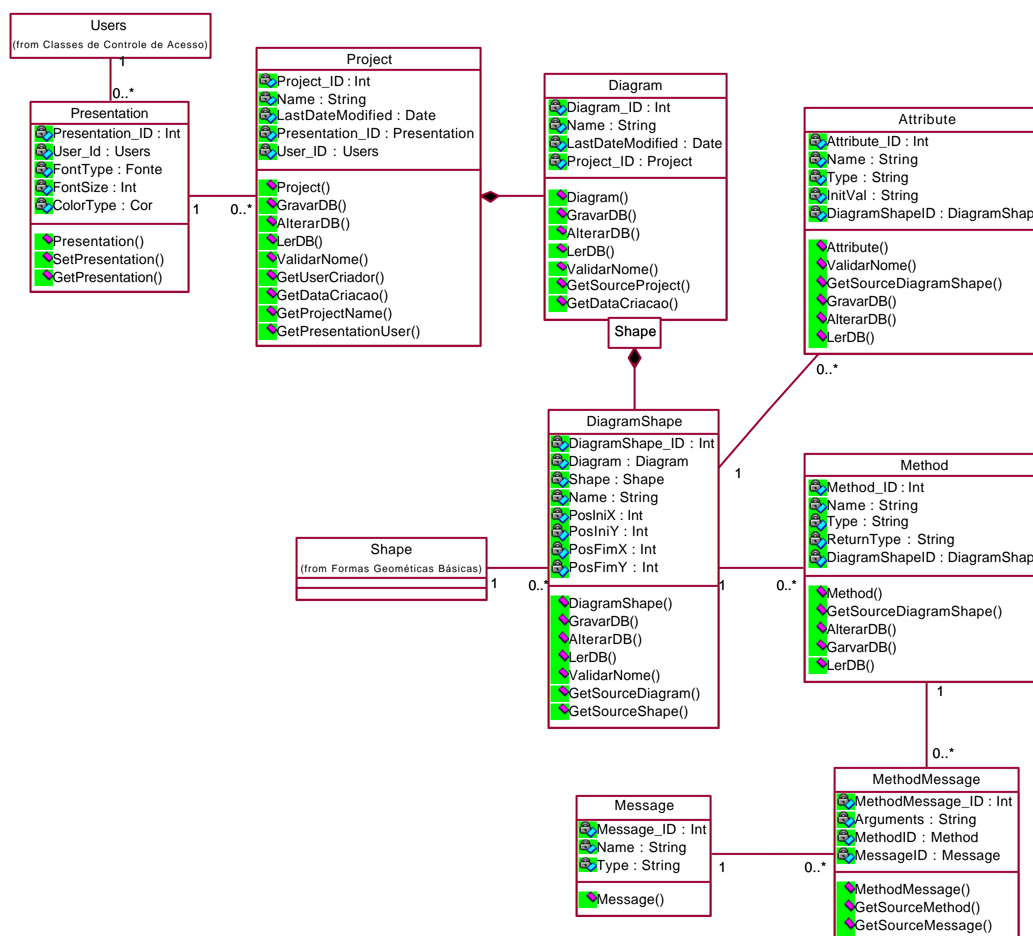


FIGURA 4.7 – Diagrama de Classes do Pacote Editor Distribuído.

[4] consultar http://planeta.terra.com.br/informatica/cayres para efetuar *download* das classes implementadas.

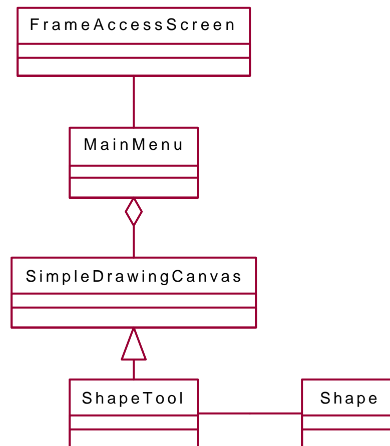


FIGURA 4.8 – Modelo de Classes da Arquitetura para Editores Distribuídos de Manipulação Colaborativa de Documentos Diagramáticos.

A seguir, serão descritas as principais classes modeladas para a arquitetura proposta no padrão UML (*Unified Modeling Language*). Tais classes não estão relacionadas com os dados a serem armazenados para o modelo de documento em edição, devendo o projetista da ferramenta preocupar-se em definir as estruturas de dados a serem utilizadas. Nelas estão contidos apenas os métodos de controle de edição e gravação de documentos em um banco de dados. Especializações nas classes da proposta de arquitetura utilizadas para representação gráfica de objetos para novos tipos de documentos resultarão na reestruturação ou criação de novas classes persistentes para armazenamento de informações sobre os elementos que comporão novas instâncias do editor.

4.6.2.1 *FrameAccessScreen*

Quando um usuário desejar acessar o editor, ele o fará através de uma URL utilizando um *browser* para navegação via *Internet*. Esta URL irá permitir acesso a uma tela para identificação inicial do usuário, devendo este informar seu *username* e *password*. Estes dados servirão para registro do usuário no sistema e verificação se este está autorizado a utilizar a ferramenta. A assinatura digital será feita através de acesso remoto a um banco de dados localizado em um provedor através de *servlets* Java, onde se encontra hospedado a instância do

FreDoc, um ambiente de edição distribuída para um determinado tipo de documento. Este processo irá retornar todas as caraterísticas do ambiente de edição definidas para o usuário em questão. Para validação deste processo de autenticação do usuário tem-se o protótipo[5] descrito no capítulo 5, instância do *framework* para Edição Distribuída de Geração Colaborativa de Documentos, o FreDoc. Durante o processo de acesso ao editor, via um objeto instanciado da Classe *FrameAccessScreen*, é que serão definidas às características de edição do usuário. A partir desta classe o usuário terá a sua disposição os recursos de edição disponíveis para este protótipo. Esta é a classe do editor que será responsável por disponibilizar os recursos de edição de diagramas, caso o usuário tenha permissão de acesso. O código Java resultante da implementação da classe *FrameAccessScreen* encontra-se no anexo A.

4.6.2.2 *MainMenu*

Caso, através do *applet FrameAccessScreen*, o usuário seja autorizado a trabalhar com o editor, o ambiente de edição colaborativa será disponibilizado através da classe *MainMenu*. Esta classe disponibilizará para o usuário acesso aos recursos de controle de edição de diagramas.

Tais recursos são: editar um novo documento diagramático; salvar documentos editados; imprimir documentos editados; abrir documentos editados; editar objetos gráficos (copiar, colar, mover, deletar); editar especificações dos objetos gráficos do documento (classes, notas e relacionamentos); geração de especificações em XML dos documentos em edição e; ajuda sobre o editor. Nesta classe são armazenadas também as informações sobre o usuário. Estas informações são as cores, tamanho e estilo da fonte, juntamente com o seu nome utilizado na identificação na barra de títulos da instância de menu. Estas informações são necessárias para definir o *layout* para visualização do diagrama pelo usuário.

4.6.2.3 *SimpleDrawingCanvas*

Esta classe é responsável por gerenciar e auxiliar no controle de edição de objetos solicitados através da classe *MainMenu* descrita na seção 4.6.2.2. Suas principais características são: identificação e execução da ação de edição solicitada pelo usuário; identificação de um objeto selecionado para edição; métodos de controle de inserção de figuras, linhas, texto; métodos que gerenciam e identificam qual ferramenta está em operação num dado instante para

[5] disponível para download em http://planeta.terra.com.br/informatica/cayres/mestrado

propiciar os recursos desta para o usuário; métodos para controlar os objetos *Vector*, responsáveis pelo armazenamento dos objetos gráficos e texto a eles relacionados durante a diagramação. O código Java resultante da implementação da classe *SimpleDrawingCanvas* encontra-se no anexo B.

Destaca-se que a classe *SimpleDrawingCanvas* é parte integrante do *framework HotDraw for Java* [KNO 97]. Adaptações feitas em sua estrutura básica possibilitaram sua utilização neste projeto. Todas as informações de um diagrama em edição vão sendo armazenadas em estruturas de dados do tipo *Vector* para posterior visualização dos objetos gráficos desenhados juntamente com o texto descrito para a especificação deste objeto. Estão sendo armazenados em estruturas diferentes (texto e figuras), visando facilitar e agilizar o processo de edição, pois, através de posições correspondentes, é possível associar o texto à figura quando necessário. O controle de atualização de visões (alterações no diagrama) é percebido e executado por esta classe também.

As demais classes que foram utilizadas para tornar possível o desenvolvimento desta proposta de arquitetura fazem parte do pacote de classes do JDK1.3 ou do *framework HotDraw for Java*, citados anteriormente no início deste capítulo.

4.7 Camadas da Arquitetura do FreDoc

O ambiente proposto para o FreDoc possui as seguintes camadas: *Database Server, User* e *Web Server*, sendo esta última camada implementada com o recurso de *Servlets* Java. A seguir, são apresentadas as características de cada uma das camadas:

DataBase Server: servidor de banco de dados onde serão armazenadas as informações de documentos construídos através do FreDoc, baseados nos modelos semânticos das classes responsáveis pelo armazenamento de informações sobre o documento em edição e de classes responsáveis pelas informações sobre usuários do sistemas (suas característica proprietárias de edição);

Web Server: servidor WWW onde estão armazenadas as páginas HTML, os *applets* e *frames* Java do ambiente gráfico de edição distribuída de documentos, juntamente com a definição das classes persistentes responsáveis pelo armazenamento local, estação de trabalho do usuário, do documento em processo de colaboração.

Users: são os usuários, proprietários e colaboradores, dos documentos editados, responsáveis pela utilização da *interface* GUI de acesso remoto ao editor que encontra-se na camado *Web Server.*

O usuário acessa o servidor *Web* (*Web Server)* na *Universal Resource Locator* (URL) do ambiente. Ao carregar a página HTML também será carregado o *Applet* Java, que consiste em uma tela de verificação de usuários que, posteriormente irá carregar o *Frame* Java para a instância implementada do FreDoc.

4.7.1 Considerações sobre a utilização de padrões projeto e arquiteturas de aplicações web na arquitetura proposta

O FreDoc faz uso extensivo dos padrões de projeto citados no capítulo 4 para tornar possível a edição de documentos diagramáticos. Utilizou-se, também, o padrão de arquitetura para *web Thick Web Client* [CON 2000] descrito na seção 3.2.2, visando a distribuição de tarefas a serem processadas, entre o cliente e servidor.

Os padrões citados foram utilizados para possibilitar a construção de editores distribuídos de manipulação colaborativa, buscando auxiliar o processo de edição e composição de objetos gráficos. O padrão *Observer* [GAM 95] foi utilizado para manter visões de modelos em edição atualizadas, através da notificação de dependentes de um componente da mudança de seu estado. Já o *Composite* [GAM 95] foi utilizado para definir elementos gráficos utilizados na composição de uma instância de editores distribuídos de manipulação colaborativa. O padrão de arquitetura para Web *Thick Web Client* [CON 2000], possibilitou a distribuição de controles de edição para o lado cliente, deixando a cargo do servidor a centralização dos dados de documentos e o controle de acesso ao ambiente de edição colaborativa.

Assim, num diagrama de classes, por exemplo, uma classe pode ser graficamente representada por um conjunto de outros elementos (textuais e gráficos). Cada atributo ou métodos pode ser um objeto gráfico único, cujo método de desenho apenas escreva a sua descrição, o mesmo acontecendo para a representação gráfica de uma classe. Desta forma torna-se necessária a utilização destes padrões para compor a forma gráfica de visualização e atualizar visões do modelo para o usuário, durante a edição de um novo objeto do tipo classe em um diagrama. O

capítulo 5 faz uso do *framework* FreDoc visando ilustrar a sua utilização na definição de uma ferramenta de edição distribuída de documentos diagramáticos, a FreDocUML. A figura 4.9 ilustra o processo de análise de informações gráficas e textuais, armazenadas para um objeto do tipo classe, editado em uma possível instância do FreDoc que disponibiliza a edição de diagramas de classes da UML. Neste caso, a classe *SimpleDrawingCanvas*, implementada na estrutura base do FreDoc, utiliza-se de seus mecanismos para realizar a junção, gráfica e textual do objeto em edição, e posterior visualização, em seu ambiente gráfico de edição de documentos diagramáticos.

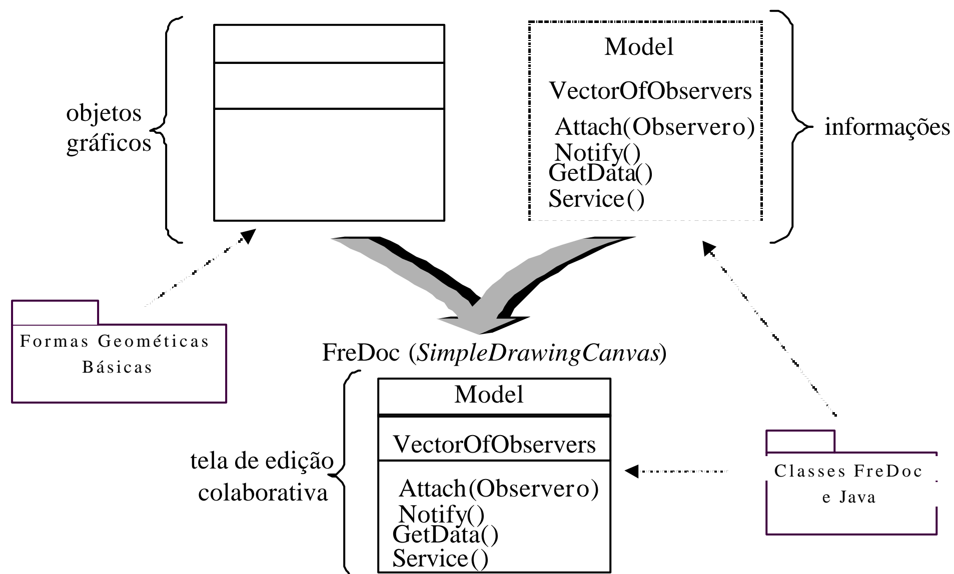


FIGURA 4.9 – Mecanismos de controle de edição de objetos gráficos do FreDoc.

4.8 Instanciando um Editor através do Framework FreDoc

Pode-se, então, pensar em se utilizar o FreDoc como base a instancia de outros tipos de documentos diagramáticos. Para tanto torna-se necessária a realização de adaptações em sua estrutura básica de classes, através do processo de herança, onde serão tratadas as especificidades das novas ferramentas de edição e dos seus objetos gráficos a serem editados.

As classes que deverão servir de base neste processo são: *SimpleDrawingCanvas*, *ShapeTool* e *Shape*. A primeira é responsável por disponibilizar o controle sobre os objetos em edição, sendo sua funcionalidade descrita no item 4.6.2.3. A classe *ShapeTool* é responsável por

controlar as funções de edição de um determinado objeto gráfico, definido pelo projetista da instancia de editor diagramático com base no FreDoc. Nela se especifica, funcionalidades como seleção de um objeto, associação de objeto gráfico com suas informações textuais, a serem armazenadas em banco de dados, redimensionamento, ligação com outros objetos gráficos (um retângulo com linhas, por exemplo), dentro outras. Vale ressaltar que deverá ocorrer uma instância desta classe para cada objeto gráfico da nova ferramenta de edição baseada no FreDoc. A última classe, *Shape*, deverá ser utilizada como base para a definição das formar geométricas de cada um dos objetos gráficos. Por exemplo, no estudo de caso apresentado no capítulo 5, esta foi utilizada como base para a definição da classe *ClassShape*, onde aplicou-se a forma geométrica de um polígono para a representação gráfica da classe durante o processo de edição.

Outro ponto importante é que fica a critério do projetista a definição de classes que correspondem aos dados que serão armazenados de cada objeto gráfico, sendo estes associados à sua forma gráfica durante o processo de edição do mesmo. Sendo assim, o projetista tem a liberdade de criar separadamente as classes que corresponderão à interface e aos dados a serem editados que, posteriormente, serão visualizados com um único objeto, conforme ilustra a figura 4.9.

Caso seja necessária a gravação de dados dos objetos editados em banco de dados, os scripts de SQL para os bancos serão enviados para o servidor, onde se encontra hospedado o editor diagramático, através da classe *FrameAccessScreen* (veja código fonte no anexo A). Esta classe foi definida como um applet (cliente) interligado a servlets (servidor) que executam o SQL solicitado pela aplicação.

5 Um estudo de caso: um editor distribuído de geração colaborativa de diagrama de classes UML

Neste capítulo são apresentadas as características de implementação do protótipo da proposta de arquitetura para ambiente distribuído de edição colaborativa descrita no capítulo 4. Este Editor foi baseado no EDI (Editor de Diagramas para a Internet) desenvolvido e apresentado por Pompermaier [POM 99] como dissertação de mestrado CPGCC em 1999. Através de estudos feitos no EDI, verificou-se que sua principal característica é proporcionar a criação e modelagem de diagrama de classes, tendo como base um ambiente distribuído. Segundo [POM 99], o EDI foi desenvolvido como uma *applet* ("carregada no cliente" assim que o mesmo acesse o editor via *Internet*), e uma aplicação executada no servidor (de onde são gerenciados os clientes e feitas as atualizações dos modelos que estão sendo desenvolvidos). Seu modelo fornece uma estrutura semântica que facilita a descrição dos conceitos e das notações utilizadas pelo *software* EDI. Desta forma, pode-se pensar em um ambiente onde uma ferramenta pode ser utilizada simultaneamente por diversos autores, com capacidade para gerenciar e atualizar visões em uma Base de Dados. Estudos feitos no EDI proporcionou o desenvolvimento de uma nova versão para este tipo de ferramenta de edição de documentos diagramáticos, implementada como uma instância do *framework* FreDoc para diagramas de classes notação UML, o FreDocUML, descrito neste capítulo.

O protótipo implementado apresenta os requisitos identificados na seção 4.6, propiciando: 1) acesso retomo de usuários; 2) definição, pelo usuário, de características visuais proprietárias; 3) geração de documentos XML sobre dados de documentos armazenados ou em edição; 4) ferramentas que possibilitam a edição objetos gráficos e textuais em documentos; e 5) ferramentas de controle de edição.

5.1 Características Gerais

O FreDocUML, ver hierarquia de classes na figura 5.1, foi desenvolvido na linguagem java [COR 96] e [FLA 97], utilizando arquitetura em camadas. Nesta arquitetura, as *applets* e *frames* java, com seus próprios métodos e objetos, criam a idéia de sistemas cliente/servidor. Teoricamente, uma *applet/frame* java pode ser uma interface de regra de negócio, GUI (*Graphical User Interface*) ou DBMS (*DataBase Manager System*), podendo ser considerada como camada. A ferramenta terá a estrutura de um *framework* MVC (Capítulo 2), e conterá três tipos fundamentais de componentes: um componente de apresentação (que contém a lógica que apresenta a informação a uma fonte externa e obtém entradas daquela fonte); um

componente de negócio (que contém a lógica da aplicação que gerencia as funções de negócios e os processos executados pela aplicação); e um componente de acesso a dados (que contém a lógica que fornece à interface um sistema de armazenamento de dados).

Para facilitar a identificação dos colaboradores, definiu-se também uma forma de apresentação para o usuário, podendo este especificar o estilo desejado para visualizar documentos diagramáticos, informando, para tanto, a cor, o estilo e o tamanho de fonte desejado para edição. Outro ponto importante a ser tratado é a extração de informações sobre o documento em edição e o posterior armazenamento destas em Bancos de Dados. Este processo irá obter os dados do modelo separadamente da visão, para possibilitar a visualização gráfica diferenciada pelos clientes. As seções a seguir descrevem o protótipo FreDocUML.

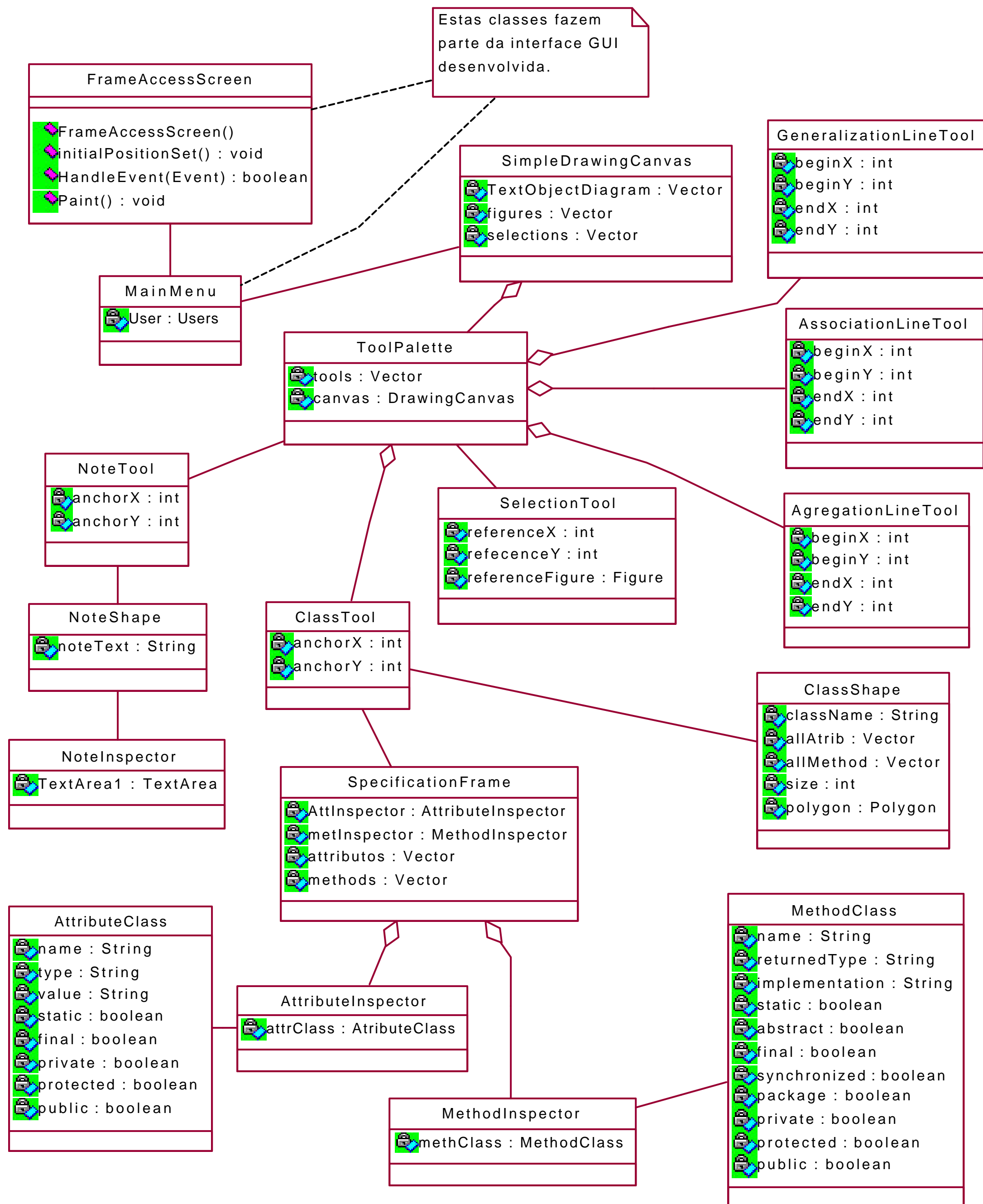


FIGURA 5.1 – Hierarquia de classes implementada para o Editor Distribuído de Manipulação Colaborativa de Documentos para diagramas de classes da UML.

5.2 Implementação do FreDocUML

O FreDocUML possui vários níveis de funcionamento. Basicamente ele pode ser dividido em um *applet* (GUI), que pode ser invocado pelo cliente através de uma URL. Este *applet* irá carregar no cliente um *framework* para edição de diagramas de classes. Esta *interface* foi desenvolvida preocupando-se com os requisitos de portabilidade, possibilitando, assim, edição de documentos através de ferramentas de controle de edição de objetos gráficos disponíveis neste protótipo além da definição de características proprietárias dos usuários colaboradores.

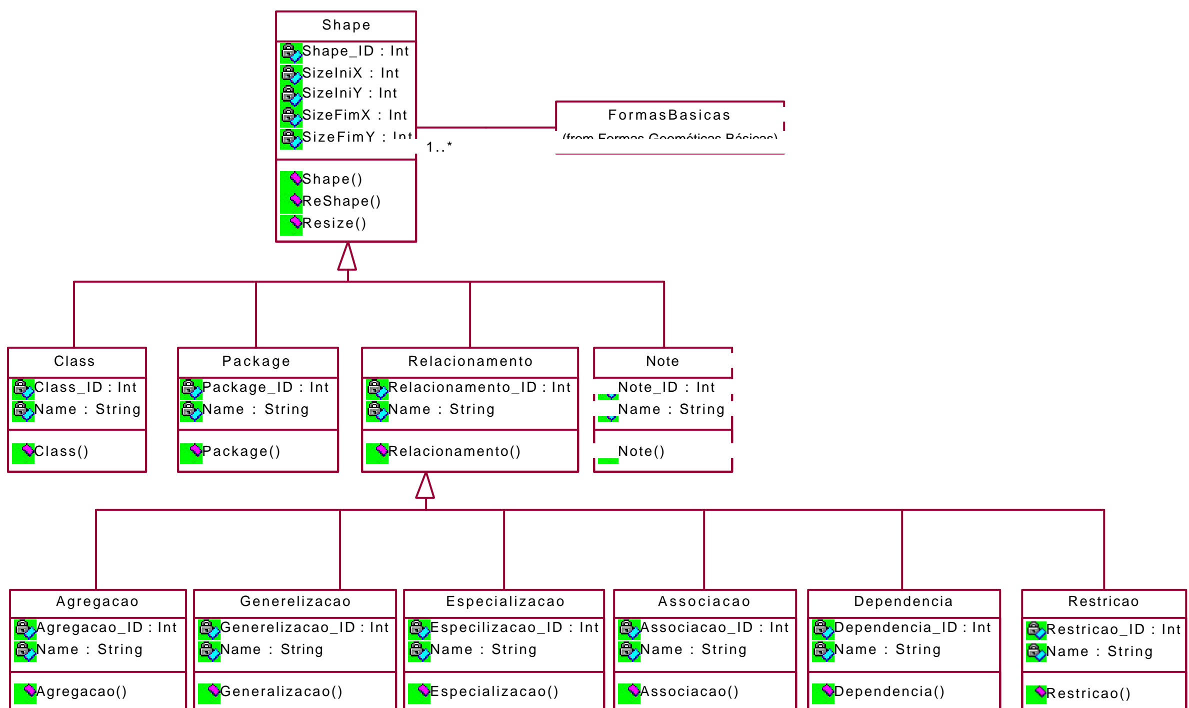


FIGURA 5.2 – Diagrama de classes do pacote *shapes* criado para apresentação gráfica dos objetos durante o processo de edição.

Desta forma, pode-se ter vários clientes trabalhando na edição de documentos diagramáticos, uma vez que o *framework* é executado localmente, na estação cliente. Obteve-se como resultado, uma biblioteca de classes[6] genérica que disponibiliza os recursos de edição para esta versão do editor. Além dos pacotes de classes apresentados no Capítulo 4, criou-se também o

---

[6] Tal biblioteca de classes encontra-se implementada e disponível para *download* em: http://planeta.terra.com.br/informatica/cayres/mestrado.

pacote *shapes*, composto por classes persistentes (figura 5.2) que será utilizado no processo de edição de objetos gráficos para diagramas de classes. Tais objetos são criados através de especializações nas classes base do pacote **Formas Geométricas Básicas**, descrito no Capítulo 4. Também foram definidas classes persistentes para armazenamento das informações do modelo, possibilitando a independência entre modelo e visão, objetivo do *framework* MVC [BRA 95a] e [BRA 95b], utilizado na implementação deste protótipo.

5.3 Funcionalidade do Protótipo

O protótipo desenvolvido para validar o *framework* FreDoc, apresentado no Capítulo 4, disponível para download em http://planeta.terra.com.br/informatica/cayres/mestrado, que disponibiliza uma página HTML, ilustrada na figura 5.3, com os principais links de navegação. Nesta página são mostrados os dados do responsável pela implementação protótipo e orientador do trabalho juntamente com links para acessar o editor, para visitar a página do grupo de pesquisa ao qual este projeto pertence, o AMADEUS e um link que disponibiliza o *download* do texto da dissertação e implementação. Tem-se também um link onde o usuário interessado poderá se cadastrar para posteriormente acessar o protótipo FreDocUML.

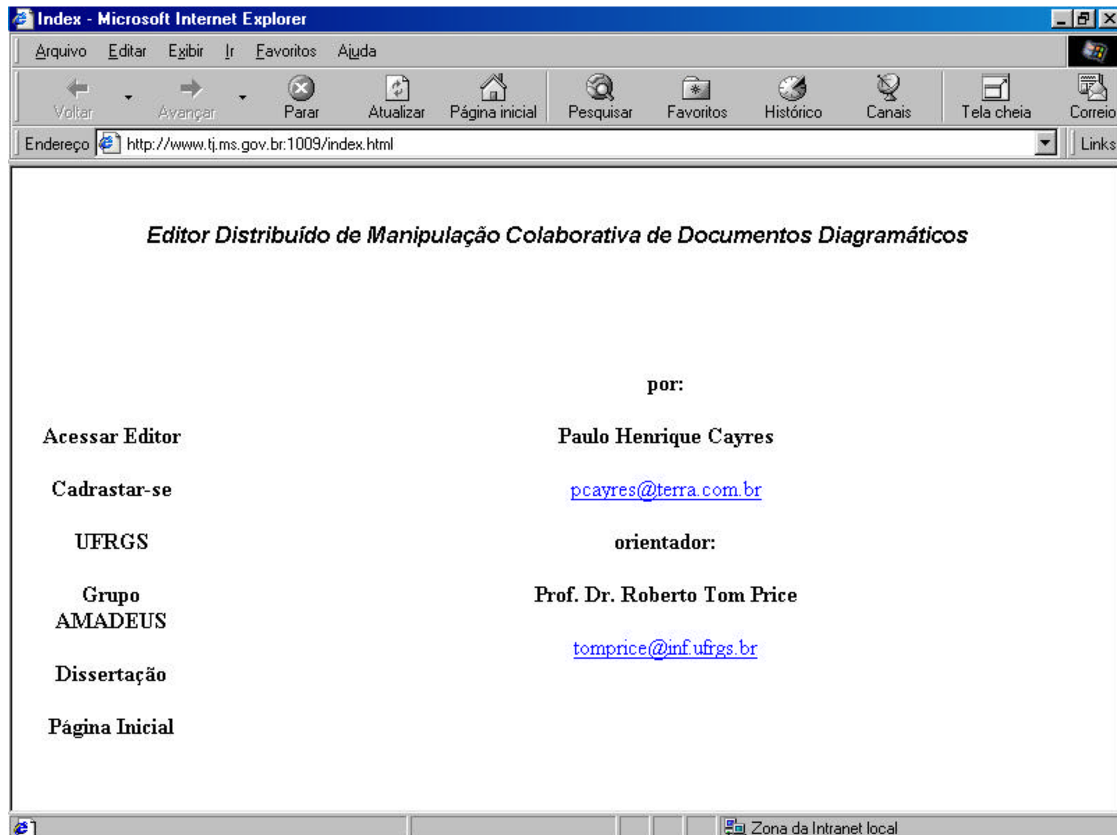


FIGURA 5.3 – Página inicial de acesso ao Editor Distribuído de Manipulação Colaborativa de Documentos Diagramáticos.

O cadastramento do usuário é feito para que se possa especificar a configuração que o mesmo deseja utilizar durante a edição de documentos diagramáticos no ambiente do FreDocUML. O formulário HTML apresentado pede algumas informações para identificação do usuário e trás uma configuração padrão de edição disponível para este protótipo. Também deverão ser informados alguns dados para controle de acesso.

FIGURA 5.4 – Página de cadastro usuários.

O processo final de cadastramento irá retornar uma página de agradecimento ao usuário, ilustrada na figura 5.5. Uma vez cadastrado, o usuário poderá acessar o ambiente de edição deste protótipo que permitirá a edição de diagramas de classes utilizando notação UML.

FIGURA 5.5 – Página de agradecimento ao usuário visitante.

5.3.1 Página para *Login* do usuário

Para inicializar a utilização do protótipo para edição colaborativa de documentos diagramáticos, o usuário deverá clicar sobre o *link* **Acessar Editor** e informar a seu *login* e senha de acesso. A página de acesso ao editor possui um *Applet* utilizado para postar informações a URL de origem do FreDocUML. Estas informações serão recebidas por um *servlet* java, responsável por autenticar e retornar ao editor às características visuais de edição de um determinado usuário. A figura 5.6 ilustra a página de acesso para FreDocUML.

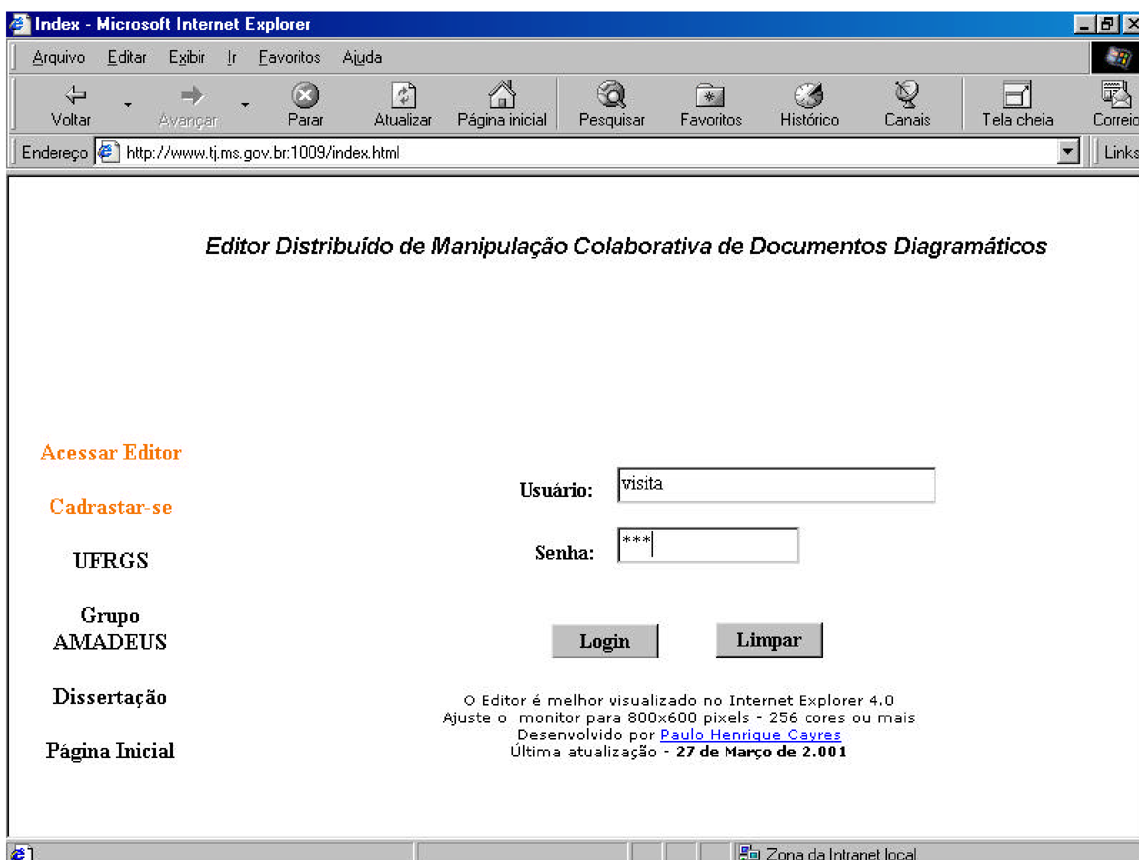


FIGURA 5.6 – Página para *Login* do usuário para o Editor Distribuído de Manipulação Colaborativa de Documentos Diagramáticos.

Após a validação do usuário será inicializado o *menu* principal do Editor. Neste *menu* estão disponíveis todas as opções para edição de diagramas. Esta versão do editor disponibiliza para o usuário a edição de diagrama de classes, baseado no modelo de classes da UML. A figura 5.7 ilustra o *menu* principal do FreDocUML. Através da opção *File / New / Class Diagram* será

possível editar diagramas de classes. Na tela de edição de diagramas, ilustrada na figura 5.8, serão disponibilizadas opções para inserção de classes, linhas de relacionamento entre classes, linhas de agregação e generalização entre classes e notas, além de opções de edição tais como: seleção, edição e movimentação de objetos gráficos; barras de rolagem vertical e horizontal e teclas de atalho para execução das principais funções de edição de objetos gráficos.

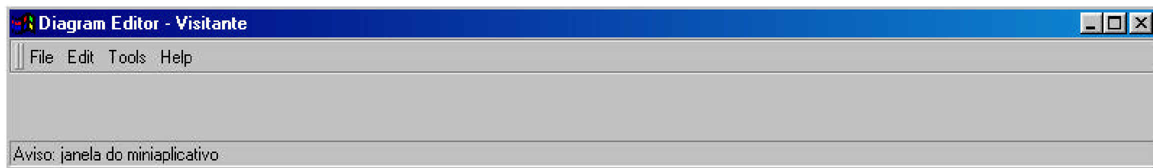


FIGURA 5.7 – Menu Principal do FreDocUML.

O menu ainda apresenta as opções *Edit*, *Tools* e *Help*. A opção *Edit* contém as ferramentas básicas de edição (*Cut*, *Delete*, *Copy*, *Paste*) para manipulação das formas gráficas utilizadas durante a edição de diagramas de classes. A opção *Tools* contém as ferramentas básicas de manipulação de informações sobre classes e notas e uma ferramenta de geração de código XML do documento diagramático em edição. A opção *Help* apresenta uma ajuda sobre o funcionamento do editor.

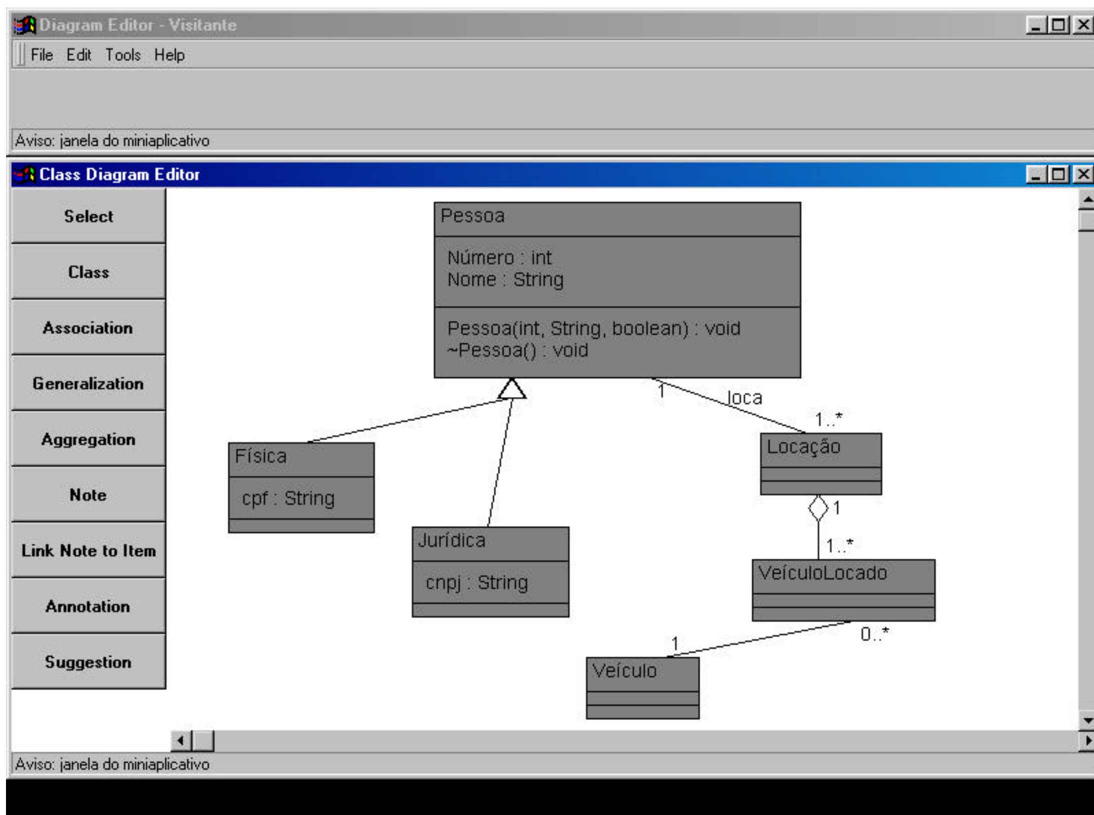


FIGURA 5.8 – Tela de edição de diagramas de classes (UML) disponível para esta versão do FreDocUML.

Selecionando um dos objetos gráficos inseridos na tela de edição de diagramas ter-se-á acesso a suas especificações através da opção *Tools/Specifications* ou através da tecla de atalho Ctrl+P. Neste instante o editor irá verificar qual o objeto que está entrando em processo de edição e disponibilizará para o usuário o *frame* que contém os recursos para que sejam informados os dados que irão compor este objeto gráfico no modelo do diagrama em edição. A figura 5.9 ilustra a especificação de modelagem para um objeto do tipo classe. O FreDocUML irá atualizar a visão do documento em edição, redimensionando os objetos gráficos para que os mesmo comportem suas informações, após o término da descrição dos dados de um objetos gráfico, identificado com a finalização da execução do *frame* por parte do usuário. O protótipo FreDocUML disponibiliza, nesta versão, a edição de objetos gráficos que compõem a estrutura de um diagrama de classes baseado no modelo de classes da UML.

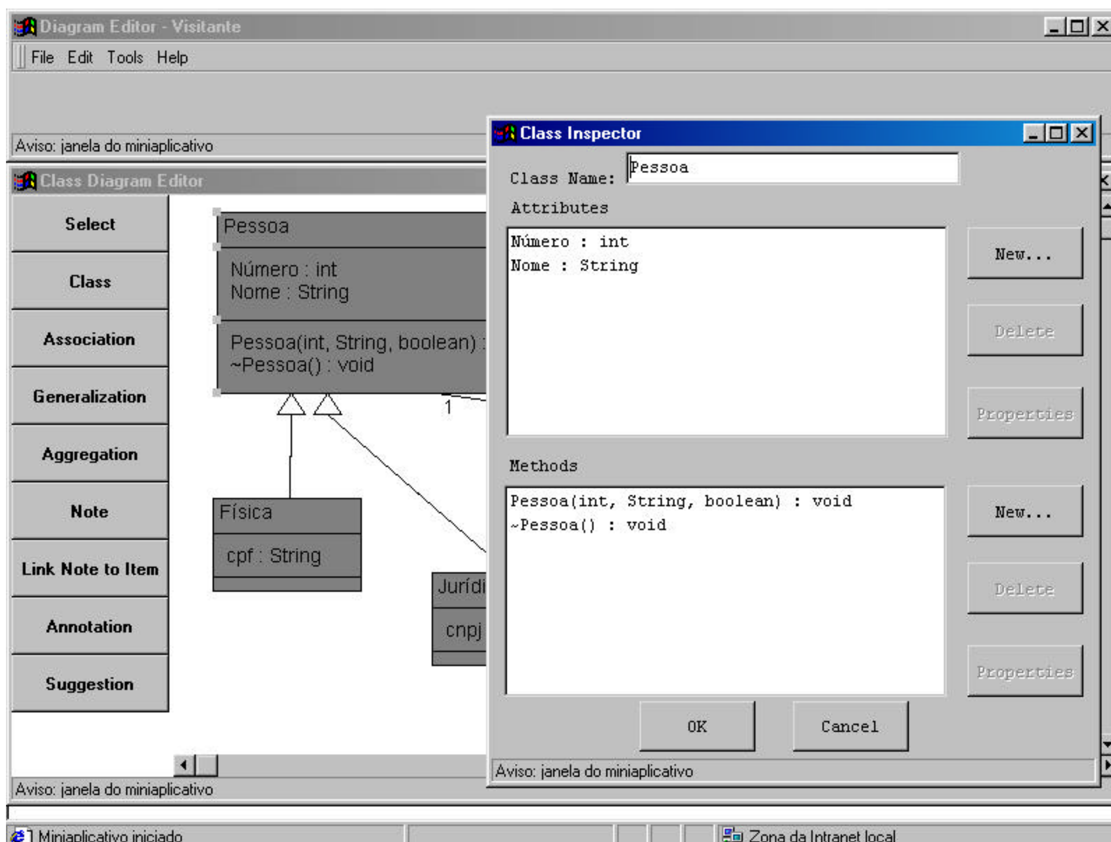


FIGURA 5.9 – Especificação de atributos e métodos para um objeto do tipo classe para o diagrama em edição.

O FreDocUML também disponibiliza uma ferramenta auxiliar de colaboração através da geração de código XML do documento diagramático em edição. Esta ferramenta encontra-se na opção *Tools/Generate XML...* do menu principal e, através da leitura das especificações editadas para os objetos do documento em edição, gera o código XML correspondente. Esta ferramenta ainda disponibiliza a edição do código gerado para que o usuário possa acrescentar informações caso queira. Ela também disponibiliza recursos de formatação do texto XML em edição, possibilitando a alteração de tipo, estilo e tamanho de fonte.

Uma última opção disponível para esta ferramenta é a possibilidade de salvar o código XML editado para posterior envio a um colaborador, por exemplo. A figura 5.10 ilustra a ferramenta auxiliar de geração de código XML.

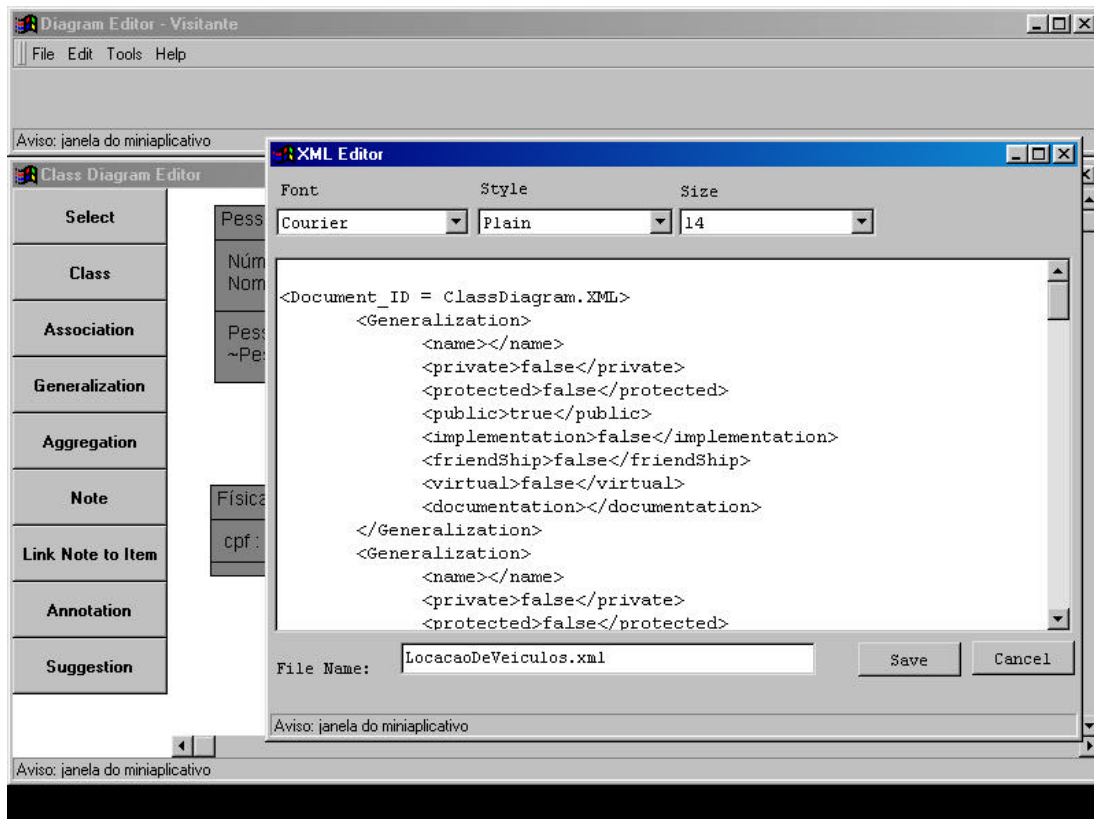


FIGURA 5.10 – Geração de documentos XML com as definições do diagrama de classes editado no FreDocUML.

Uma outra forma de colaboração é a inserção de sugestões de mudanças no documento em edição. Para que isto ocorra, o usuário deverá selecionar um objeto gráfico do documento diagramático em edição que deseja inserir uma sugestão e, depois, clicar no botão *Suggestion*, que se encontra na barra de ferramentas vertical ao lado da área de edição, como ilustrado na figura 5.11. Nesta versão do FreDocUML a ferramenta de sugestão está disponível apenas para objetos do tipo classe, utilizados para construção de diagramas de classes, baseados no modelo de classes da UML. As sugestões possíveis são: 1) apenas um texto informacional; 2) a exclusão de um atributo ou método desta classe; e 3) o deslocamento de um atributo ou método para uma outra classe. Ressalta-se que, nesta versão, não está sendo feita uma análise da possibilidade do envio de um atributo ou método para uma outra classe, visando consistência do documento.

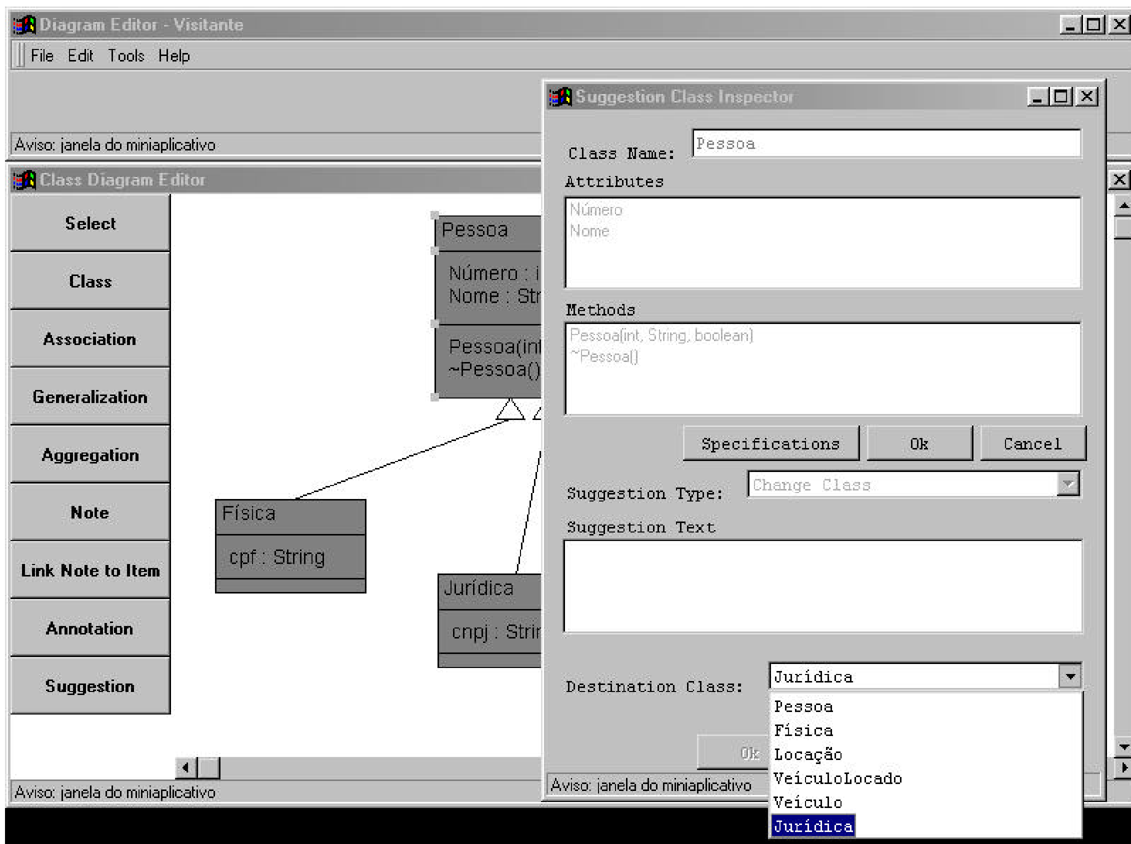


FIGURA 5.11 – Colaboração através da inserção de sugestões na especificação de um objeto do tipo classe.

O FreDocUML disponibiliza, ainda nesta versão, a possibilidade de salvar e abrir documentos diagramáticos. Ambas as opções disponibilizam uma lista dos arquivos já gerados e existentes no bando de dados modelado para este protótipo, para que o usuário possa escolher qual arquivo deseja abrir ou, simplesmente, gravar um novo arquivo. Em ambos os casos, o usuário poderá visualizar uma lista com todos os documentos existentes ou uma lista apenas com documentos por ele gerados. A figura 5.12 ilustra a tela de salvamento de documentos diagramáticos editados. Neste exemplo estão sendo listados apenas os arquivos gerados pelo usuário em questão.

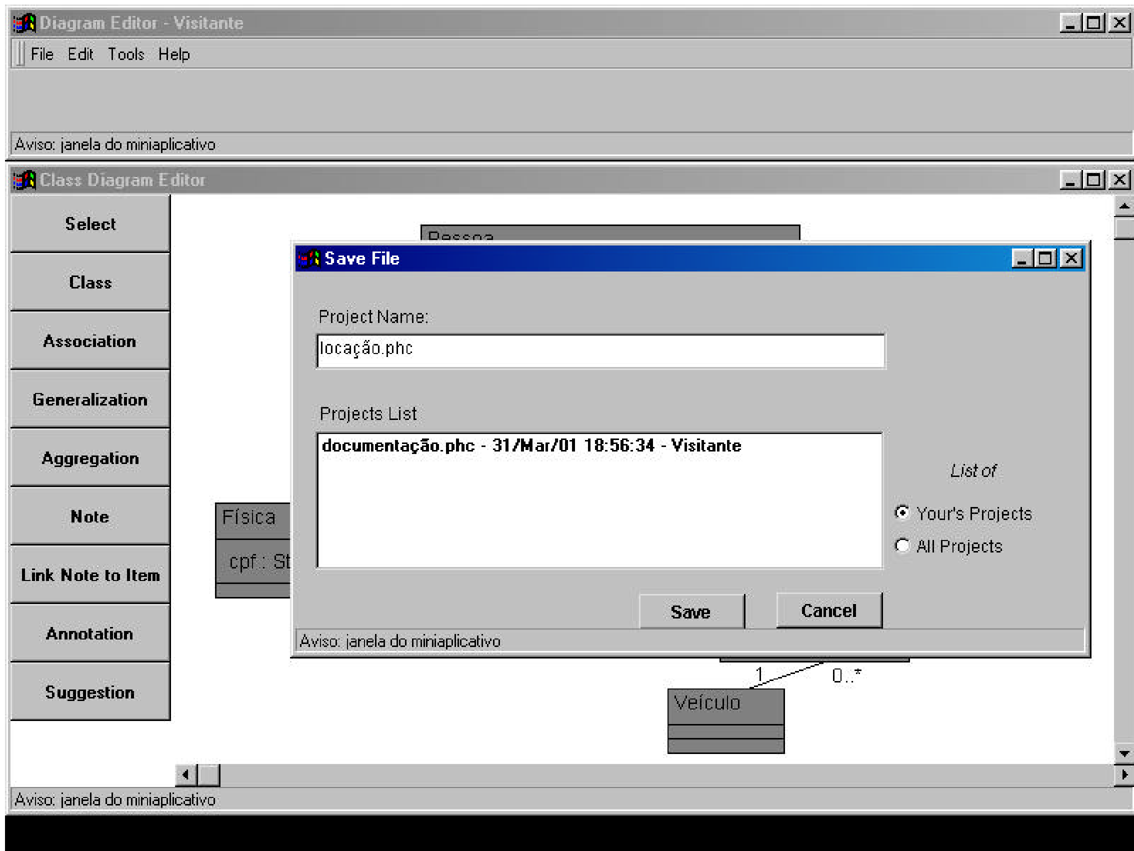


FIGURA 5.12 – Salvando um documento diagramático editado pelo FreDocUML.

Já a figura 5.13 ilustra a tela de abertura de documentos existentes no banco de dados do protótipo FreDocUML especificamente, neste exemplo, listando todos os documentos já gravados. O *CheckBox* intitulado *List of* permite ao usuário visualizar apenas os arquivos por ele editados.

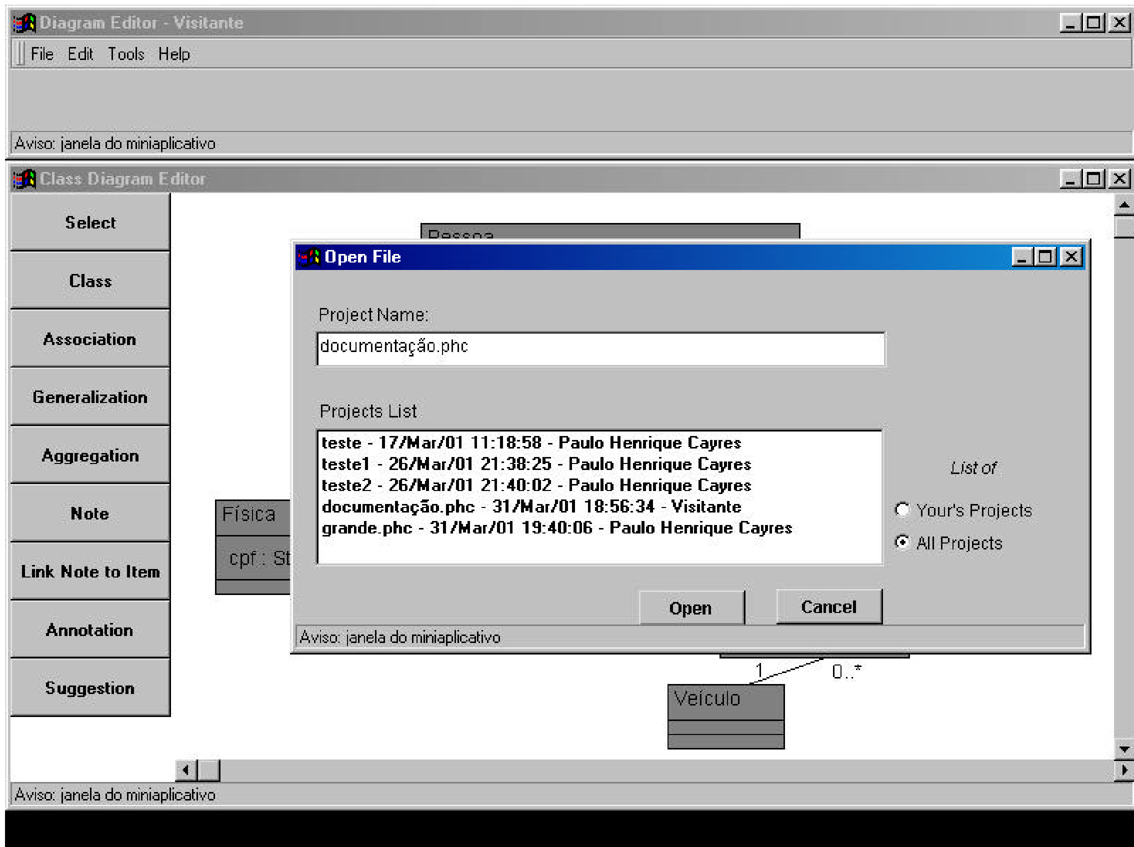


FIGURA 5.13 – Abrindo um documento diagramático editado pelo FreDocUML.

A página principal de acesso do FreDocUML também disponibiliza uma ferramenta de envio de *e-mails*, caso o usuário queira enviar alguma mensagem sem a necessidade da utilização de um *WebMail* configurado. A figura 5.14 mostra a tela de envio de e-mail. Nesta versão está sendo disponibilizado apenas o envio de mensagens, sem a possibilidade de arquivo em anexo.

Index - Microsoft Internet Explorer

Arquivo Editar Exibir Ir Favoritos Ajuda

Voltar Avançar Parar Atualizar Página inicial Pesquisar Favoritos Histórico Canais Tela cheia Correio

Endereço http://www.tj.ms.gov.br:1009/index.html Links

Editor Distribuído de Manipulação Colaborativa de Documentos Diagramáticos

Acessar Editor

Cadrastar-se

UFRGS

Grupo AMADEUS

Dissertação

E-Mail

Página Inicial

De : (Nome)

De : (email)

Para: (Nome)

Para: (email)

Assunto:

Mensagem:

Enviar Limpar

Zona da Internet

FIGURA 5.14 – Ferramenta auxiliar para envio de *e-mail* do FreDocUML.

6 Conclusão

Nos capítulos anteriores apresentou os conceitos necessários para o desenvolvimento de um *framework* a ser utilizado como base para criação de editores distribuídos de manipulação colaborativa de documentos diagramáticos. Percebeu-se, ao longo deste trabalho, que a principal utilidade de um editor com estas especificações, é a disponibilização de recursos para um processo de intercâmbio de conhecimento durante a edição de um documento. Neste sentido, o principal objetivo deste trabalho foi à especificação de um *framework* que tornasse possível a construção de tais editores.

Entretanto, a dificuldade de associação de um modelo que sirva de base para o armazenamento dos documentos compartilhados com a estrutura do *framework* FreDoc, proposto neste trabalho, torna complexo o processo de criação de editores distribuídos.

Apesar da relativa facilidade encontrada através da utilização de padrões de projetos, unida com a possibilidade de reutilização de código disponível pela orientação a objetos, destaca-se a dificuldade na definição de mecanismos para associação de formas gráficas com sua visão no banco de dados. A utilização de padrões no processo de desenvolvimento do FreDoc auxiliou na especificação de pacotes de classes para controle de visões de um modelo e composição de objetos gráficos, largamente utilizados em ambientes gráficos de edição. Neste trabalho fez-se o uso dos padrões *Observer* e *Composite* [GAM 95] como base para a criação/edição de objetos gráficos. O padrão *Observer* foi utilizado para auxiliar o processo de atualização de visões, enquanto que o *Composite* foi empregado no processo de definição das formas gráficas a serem visualizadas durante a edição de documentos diagramáticos. Utilizou-se também o padrão de arquitetura *Thick Web Client* de aplicações para *web* na elaboração do FreDoc, buscando a distribuição de recursos de processamento entre o cliente e o servidor.

Além do uso de padrões, buscou-se uma linguagem de programação que suportasse o desenvolvimento para ambientes distribuídos. Java apresentou-se como uma boa solução por possuir em sua estrutura classes que serviram de base para o desenvolvimento de um protótipo, o FreDocUML, buscando validar o *framework* FreDoc. O processo de prototipação auxiliou também no aperfeiçoamento do FreDoc, ao longo de sua implementação.

Por fim, especificou-se mecanismos de colaboração entre os usuários deste editor para torná-lo colaborativo. A solução apresentada pelo FreDoc baseia-se no modelo proposto por Souza [SOU 97], que disponibiliza o recurso de anotação dentro do documento colaborado. Também foi implementado o serviço de *e-mail* como outra forma de comunicação. O uso deste modelo permitiu a criação de um ambiente de edição amigável para possibilitar a colaboração durante o processo de edição de um documento diagramático.

6.1 Contribuições

O trabalho de pesquisa apresentado produziu um conjunto de resultados que engloba os objetivos estabelecidos no capítulo de introdução do mesmo. As principais contribuições estão descritas a seguir.

Proposta de uma arquitetura de framework para ambientes de edição colaborativa de documentos diagramáticos. Esta arquitetura se caracteriza pela flexibilidade para definição de novos ambientes de edição colaborativa, cabendo ao projetista especificar os novos componentes gráficos de edição a serem utilizados, tendo como base às classes do FreDoc.

A utilização de colaboração durante o processo de edição de documentos está disponível através de anotações, inserção de novos componentes gráficos em um documento colaborado, geração de *scripts* XML de documentos em edição ou que já tenham sido armazenados em banco de dados e ferramenta para envio de e-mail entre os colaboradores.

Geração de versões de um documento, através da gravação de colaborações sugeridas para documentos em arquivos separados do arquivo original, sendo identificado o autor da colaboração.

Implementação e documentação de um protótipo para validar alguns dos requisitos da arquitetura FreDoc, descritas na seção 4.5.

6.2 Trabalhos Futuros

Ao término deste trabalho percebeu-se que a arquitetura do FreDoc poderia ser estendida, visando o acréscimo de ferramentas auxiliares que estão descritas nas próximas seções.

6.2.1 Suporte a diferentes idiomas

Anexar à ferramenta mecanismos para que o usuário possa selecionar o idioma desejado para trabalhar com o editor. Tal recurso poderia ser implementado através da inserção de mais uma característica às propriedades do usuário. Durante o processo de acesso, uma busca seria realizada em um banco de idiomas (gravadas em um banco de dados) para que a *interface* gráfica seja carregada através do *browser* com tal definição.

6.2.2 Geradores de código

Gerar código automaticamente após ter sido finalizado o processo de edição do documento. Para tanto, deve-se realizar um estudo prévio para definição das linguagens a serem utilizadas (*C++, Java, SmallTalk*) como base para o código a ser gerado, juntamente com uma análise das classes persistentes utilizadas no processo de armazenamento dos dados diagramados. Pode-se trabalhar, por exemplo, com a geração de código estrutural, onde são gerados "esqueletos" de programas a partir das especificações de um modelo. Este processo deve ser seguido de uma fase onde o código gerado será complementado de uma forma não-automatizada.

6.2.3 Validade do modelo

Garantir a consistência semântica de especificações, localizando construções incorretas, ambigüidades e estruturas incompletas, preferencialmente promovendo correção automática ou dirigindo o processo de correção. Esta função, agregada a um ambiente colaborativo de edição de documentos, é particularmente importante, pois envolve grande quantidade de informações. Sugere-se, então, a definição de uma enciclopédia visando um controle sobre a consistência entre diagramas através de mecanismos de verificação e validação. Desta forma, a ferramenta terá uma verificação mais crítica do documento, não ficando restrita apenas à criação, diagramação e armazenamento.

6.2.4 Controle de Visão

Possibilitar que diversos usuários possam estar trabalhando com visões distintas de um mesmo documento. Os usuários deverão ser notificados caso haja alguma mudança no documento e este deverá decidir se deseja atualizar sua visão ou não. Uma sugestão seria a integração do editor distribuído com a enciclopédia para ambientes de desenvolvimento de *software*. Desta forma, pode-se trabalhar com uma atualização *on-line* do documento em edição, uma vez que a enciclopédia teria o controle sobre os documentos e seus colaboradores. Sugere-se um estudo sobre o ED-ADS, proposto recentemente por Notari [NOT 2000], e como este poderá integrar-se com um ambiente gráfico de edição de documentos.

6.2.5 Integração com outras ferramentas (importação/exportação de documentos)

Estudar a viabilidade de integração de editores baseados na arquitetura proposta com outras ferramentas de diagramação. Isto poderia ser feito através da análise da estrutura de armazenamento dos documentos editados em outras ferramentas (*Rational Rose*, por exemplo) e o desenvolvimento de um módulo de importação/exportação através da leitura, tradução e adaptação de documentos.

Anexo A Implementação da Classe *FrameAccessScreen*

```
import java.awt.*;
import java.applet.*;
import java.io.*;
import java.util.*;
import java.net.*;
import java.util.Vector;
import java.util.Enumeration;
import java.util.Hashtable;

import MainMenu;
import ksc.hotdraw.*;
// Class acesso
public class FrameAccessScreen extends Applet
{
   final int MenuBarHeight = 0;

   // Component Declaration
   public TextField TextField1;
   public TextField TextField2;
   public Button Button1;
   public Button Button2;
   public Label Label1;
   public Label Label2;

   public String driverField = "sun.jdbc.odbc.JdbcOdbcDriver";
   public String urlField = "jdbc:odbc:Editor.mdb";
   public String userField = "";
   public String passwordField = "";

   public messagebox messag;
   public messagedlg mens;
   public static MainMenu MainMenu1;
   public static Salvar save;

   // End of Component Declaration

   // init()
   public void init()
   {
      // Frame Initialization
      setForeground(Color.black);
      setBackground(Color.white);
      setFont(new Font("Dialog",Font.BOLD,14));
      setLayout(null);
      // End of Frame Initialization

      // Component Initialization
      TextField1 = new TextField("");
      TextField1.setForeground(Color.black);
      TextField1.setBackground(Color.white);
      TextField1.setFont(new Font("TimesRoman",Font.PLAIN,14));
      TextField2 = new TextField("");
      TextField2.setForeground(Color.black);
      TextField2.setBackground(Color.white);
      TextField2.setFont(new Font("TimesRoman",Font.PLAIN,14));
      Button1 = new Button("Login");
      Button1.setFont(new Font("TimesRoman",Font.BOLD,14));
      Button2 = new Button("Limpar");
      Button2.setFont(new Font("TimesRoman",Font.BOLD,14));
      Label1 = new Label("Usuário:",Label.LEFT);
      Label1.setFont(new Font("TimesRoman",Font.BOLD,14));
      Label2 = new Label("Senha:",Label.LEFT);
      Label2.setFont(new Font("TimesRoman",Font.BOLD,14));
      TextField2.setEchoCharacter('*');
      // End of Component Initialization
```

```
        // Add()s
        add(TextField1);
        add(TextField2);
        add(Label2);
        add(Label1);
        add(Button1);
        add(Button2);
        // End of Add()s

        InitialPositionSet();
    } // End of init()

    // start()
    public void start()
    {
    } // End of start()

    // stop()
    public void stop()
    {
    } // End of stop()

    // destroy()
    public void destroy()
    {
    } // End of destroy()

    public void paint(Graphics g)
    {
        // paint()
        // End of paint()
    }

    public void InitialPositionSet()
    {
        // InitialPositionSet()
        resize(392,273);
        TextField1.reshape(136,96+MenuBarHeight,224,26);
        TextField2.reshape(136,138+MenuBarHeight,128,26);
        Button1.reshape(90,206+MenuBarHeight,75,25);
        Button2.reshape(205,205+MenuBarHeight,75,25);
        Label1.reshape(66,102+MenuBarHeight,69,19);
        Label2.reshape(76,146+MenuBarHeight,57,19);
        TextField1.requestFocus();
        // End of InitialPositionSet()
    }

    public boolean handleEvent(Event evt)
    {
        // handleEvent()
        if (evt.id == Event.ACTION_EVENT && evt.target == Button1) Button1_Action(evt.target);
        else if (evt.id == Event.ACTION_EVENT && evt.target == Button2) Button2_Action(evt.target);
        // End of handleEvent()

        return super.handleEvent(evt);
    }

    // Event Handling Routines
    public void acesso_WindowDestroy(Object target)
    {
        System.exit(0);
    }

    public void Button1_Action(Object target)
    {
      if(verificar()){
//Acesso a Servlet para verificação de senha
        try {
                String protocol = getDocumentBase().getProtocol();
            String host = getDocumentBase().getHost();
```

```
            int port = getDocumentBase().getPort();

//          As duas linhas em comentario tambem funcionam
//          if (port != -1) host += ":" + port;
//          URL servletURL = new URL("http://" + host + "/servlet/AcessoAppletServlet");

            URL servletURL = new URL(protocol, host, port, "/servlet/AcessoAppletServlet");

         URLConnection uc = servletURL.openConnection();
            uc.setDoOutput(true);
            uc.setDoInput(true);
            uc.setUseCaches(false);
            uc.setRequestProperty("Content-type","application/octet-stream");


            ObjectOutputStream objOut = new ObjectOutputStream(uc.getOutputStream());

            Date now = new Date();
            String conArray[] = new String[10];
            conArray[0]  = driverField;
            conArray[1]  = urlField;
            conArray[2]  = userField;
            conArray[3]  = passwordField;
            conArray[4]  = TextField1.getText();
            conArray[5]  = TextField2.getText();
            conArray[6]  = now.toLocaleString();

            objOut.writeObject(conArray);
         objOut.flush();
         objOut.close();

// get objects from servlet (1. boolean validation)
         ObjectInputStream objIn = new ObjectInputStream  (uc.getInputStream());
            String Returntest = (String)objIn.readObject();
            String nome=(String)objIn.readObject();
            String email=(String)objIn.readObject();
            String fonte=(String)objIn.readObject();
            String tama=(String)objIn.readObject();
            int tamanho = Integer.valueOf(tama).intValue();
            String estilo=(String)objIn.readObject();
            String corfonte=(String)objIn.readObject();
            String corobjetos=(String)objIn.readObject();

        objIn.close();

            if (Returntest.equals("false")){
                  messag = new messagebox("Acesso Negado");
                  messag.show();
            }
            else {
                  MainMenu1 = new MainMenu(nome, fonte, tamanho, estilo, corfonte, corobjetos);
                  MainMenu1.show();
                  save = new Salvar(protocol, host, port);
            }
    }
    catch (Exception e) {
          System.out.println(e);
    }
     }
     else {
            messag = new messagebox("Favor informar Usu rio e Senha.");
            messag.show();
     }
     LimparCampos();
    }

    public void Button2_Action(Object target)
    {
        LimparCampos();
```

```
    }

    public void LimparCampos() {
          TextField1.setText("");
          TextField2.setText("");
          TextField1.requestFocus();
    }

    public boolean verificar(){
       if(TextField1.getText().equals("") && TextField2.getText().equals(""))
              return false;
       return true;
    }

    // End of Event Handling Routines

} // End of Class acesso

class Salvar extends Applet{

     public String driverField = "sun.jdbc.odbc.JdbcOdbcDriver";
     public String urlField = "jdbc:odbc:Editor.mdb";
     public String userField = "";
     public String passwordField = "";
     public static messagebox messag;
     public String Sprotocol;
     public String Shost;
     public int Sport;
     public messagedlg mens;

     //Construtor
     public Salvar(String protocol, String host, int port){
        Sprotocol = protocol;
        Shost = host;
        Sport = port;
     }

     public String OpenFiles(String nomeArq, String autor,String data){
       try {

              URL servletURL = new URL(Sprotocol, Shost, Sport, "/servlet/OpenProject");

              URLConnection Suc = servletURL.openConnection();
              Suc.setDoOutput(true);
              Suc.setDoInput(true);
              Suc.setUseCaches(false);
              Suc.setRequestProperty("Content-type","application/octet-stream");


              ObjectOutputStream SobjOut = new ObjectOutputStream(Suc.getOutputStream());

              String SconArray[] = new String[10];
              SconArray[0]  = driverField;
              SconArray[1]  = urlField;
              SconArray[2]  = userField;
              SconArray[3]  = passwordField;
              SconArray[4]  = autor;
              SconArray[5]  = nomeArq;
              SconArray[6]  = data;

              SobjOut.writeObject(SconArray);
              SobjOut.flush();
              SobjOut.close();

// get objects from servlet (1. boolean validation)
              ObjectInputStream SobjIn = new ObjectInputStream  (Suc.getInputStream());
              String SReturntest = (String)SobjIn.readObject();

              SobjIn.close();
              return SReturntest;
```

```
 }
catch (Exception e) {
        System.out.println(e);
        messag = new messagebox("erro ao tentar abrir arquivo.");
        messag.show();
}
     return "false";
 }

 public void SaveFiles(String nomeArq, String autor){
   try {

          URL servletURL = new URL(Sprotocol, Shost, Sport, "/servlet/SaveProject");

          URLConnection Suc =  servletURL.openConnection();
          Suc.setDoOutput(true);
          Suc.setDoInput(true);
          Suc.setUseCaches(false);
          Suc.setRequestProperty("Content-type","application/octet-stream");


          ObjectOutputStream SobjOut = new ObjectOutputStream(Suc.getOutputStream());

          Date now = new Date();
          String SconArray[] = new String[10];
          SconArray[0]  = driverField;
          SconArray[1]  = urlField;
          SconArray[2]  = userField;
          SconArray[3]  = passwordField;
          SconArray[4]  = autor;
          SconArray[5]  = nomeArq;
          SconArray[6]  = now.toLocaleString();

          SobjOut.writeObject(Sc onArray);

           SobjOut.writeObject(acesso.MainMenu1.ClassDiagramEditor1.ClassDiagramCanvas.figures);
           SobjOut.writeObject(acesso.MainMenu1.ClassDiagramEditor1.ClassDiagramCanvas.TextObjectDiagram);

          SobjOut.flush();
          SobjOut.close();

// get objects from servlet (1. boolean validation)
          ObjectInputStream SobjIn = new ObjectInputStream  (Suc.getInputStream());
          String SReturntest = (String)SobjIn.readObject();

          SobjIn.close();
 }
catch (Exception e) {
        System.out.println(e);
        messag = new messagebox("erro ao tentar Salvar arquivo.");
        messag.show();
}

 }

 public String SaveXML(String nomeArq, String XMLCode, String autor, String fonte, String estilo, String tamanho){
   try {

          URL servletURL = new URL(Sprotocol, Shost, Sport, "/servlet/SaveXML");

          URLConnection Suc = servletURL .openConnection();
          Suc.setDoOutput(true);
          Suc.setDoInput(true);
          Suc.setUseCaches(false);
          Suc.setRequestProperty("Content-type","application/octet-stream");


          ObjectOutputStream SobjOut = new ObjectOutputStream(Suc.getOutputStream());

          Date now = new Date();
```

```
            String SconArray[] = new String[15];
            SconArray[0]  = driverField;
            SconArray[1]  = urlField;
            SconArray[2]  = userField;
            SconArray[3]  = passwordField;
            SconArray[4]  = autor;
            SconArray[5]  = nomeArq;
            SconArray[6]  = XMLCode;
            SconArray[7]  = fonte;
            SconArray[8]  = estilo;
            SconArray[9]  = tamanho;
            SconArray[10]  = now.toLocaleString();

            SobjOut.writeObject(SconArray);

            SobjOut.flush();
            SobjOut.close();

// get objects from servlet  (1. boolean validation)
            ObjectInputStream SobjIn = new ObjectInputStream  (Suc.getInputStream());
            String SReturntest = (String)SobjIn.readObject();

            SobjIn.close();
            return SReturntest;
    }
   catch (Exception e) {
          System.out.println(e);
          messag = new messagebox("erro ao tentar Salvar arquivo.");
          messag.show();
   }
        return "false";
    }

    public Vector ListAllFiles(String tipo){
      try {

            URL servletURL = new URL(Sprotocol, Shost, Sport, "/servlet/ListProjects");

            URLConnection Suc = servletURL.openConnection();
            Suc.setDoOutput(true);
            Suc.setDoInput(true);
            Suc.setUseCaches(false);
            Suc.setRequestProperty("Content-type","application/octet-stream");


            ObjectOutputStream SobjOut = new ObjectOutputStream(Suc.getOutputStream());

            Date now = new Date();
            String SconArray[] = new String[10];
            SconArray[0]  = driverField;
            SconArray[1]  = urlField;
            SconArray[2]  = userField;
            SconArray[3]  = passwordField;
            SconArray[4]  = acesso.MainMenu1.Nom;
            SconArray[5]  = tipo;

            SobjOut.writeObject(SconArray);
            SobjOut.flush();
            SobjOut.close();

// get objects from servlet (1. boolean validation)
            ObjectInputStream SobjIn = new ObjectInputStream  (Suc.getInputStream());
            Vector VReturntest = (Vector)SobjIn.readObject();

            SobjIn.close();

            return VReturntest;
    }
   catch (Exception e) {
          System.out.println(e);
```

```
            messag = new messagebox("erro ao tentar executar o servlet para salvar arquivo.");
            messag.show();
        }
          return null;

     }

     public Vector FindText(String projeto, String autor, String data){
       try {

              URL servletURL = new URL(Sprotocol, Shost, Sport, "/servlet/ReturnText");

              URLConnection Suc = servletURL.openConnection();
              Suc.setDoOutput(true);
              Suc.setDoInput(true);
              Suc.setUseCaches(false);
              Suc.setRequestProperty("Content-type","application/octet-stream");


              ObjectOutputStream SobjOut = new ObjectOutputStream(Suc.getOutputStream());

              String SconArray[] = new String[10];
              SconArray[0]  = driverField;
              SconArray[1]  = urlField;
              SconArray[2]  = userField;
              SconArray[3]  = passwordField;
              SconArray[4]  = projeto;
              SconArray[5]  = autor;
              SconArray[6]  = data;

              SobjOut.writeObject(SconArray);
              SobjOut.flush();
              SobjOut.close();

// get objects from servlet (1. boolean validation)
              ObjectInputStream SobjIn = new ObjectInputStream  (Suc.getInputStream());
              Vector VReturntest = (Vector)SobjIn.readObject();

              SobjIn.close();

              return VReturntest;
     }
    catch (Exception e) {
            System.out.println(e);
            messag = new messagebox("erro ao tentar ler dados do modelo em arquivo.");
            messag.show();
    }
          return null;

     }

     public Vector FindFigures(String projeto, String autor, String data){
       try {

              URL servletURL = new URL(Sprotocol, Shost, Sport, "/servlet/ReturnFigures");

              URLConnection Suc = servletURL.openConnection();
              Suc.setDoOutput(true);
              Suc.setDoInput(true);
              Suc.setUseCaches(false);
              Suc.setRequestProperty("Content-type","application/octet-stream");


              ObjectOutputStream SobjOut = new ObjectOutputStream(Suc.getOutputStream());

              String SconArray[] = new String[10];
              SconArray[0]  = driverField;
              SconArray[1]  = urlField;
              SconArray[2]  = userField;
              SconArray[3]  = passwordField;
```

```
            SconArray[4]  = projeto;
            SconArray[5]  = autor;
            SconArray[6]  = data;

            SobjOut.writeObject(SconArray);
            SobjOut.flush();
            SobjOut.close();

// get objects from servlet (1. boolean validation)
            ObjectInputStream SobjIn = new ObjectInputStream  (Suc.getInputStream());
            Vector VReturntest = (Vector)SobjIn.readObject();

            SobjIn.close();

            return VReturntest;
    }
   catch (Exception e) {
          System.out.println(e);
          messag = new messagebox("erro ao tentar ler dados do modelo em arquivo.");
          messag.show();
   }
      return null;
    }

}
```

Anexo B Implementação da Classe *SimpleDrawingCanvas*

```
class SimpleDrawingCanvas extends java.awt.Canvas implements DrawingCanvas, FigureObserver {

/**
        * Vector utilizado para armazenar os textos referentes aos objetos
        * desenhados para este diagrama
 */
      protected Vector TextObjectDiagram = new Vector();
      protected Vector SelectedText = new Vector();

//Vector com texto dos objetos a serem copiados
      protected Vector Textcopy = new Vector();

            /**
             * The tool which is the first thing we look to in order to handle events.
             */

            protected EventHandler tool = defaultTool();

      /**
             * The figures which appear on the canvas.
             */
      protected Vector figures = new Vector();

            protected Vector selections = new Vector();
      protected Vector SelectedFigures = new  Vector();
      protected Vector handles = new Vector();
            protected Hashtable figureHandles = new Hashtable();
            protected DrawingStyle style = defaultStyle();
            static Vector clipboard = new Vector();

   /**
    * Paints the component.
    */
   public void paint(Graphics g) {
            super.paint(g);
            Rectangle clip = g.getClipRect();
            if (clip.height == -1 || clip.width == -1) {
                        paintCompletely(g);
                        return;
            }
            g.setColor(getForeground());

      int i=0;
            for (Enumeration e = figures.elements() ; e.hasMoreElements() ;) {
                        Figure fig = (Figure)e.nextElement();
                        if (fig.intersects(clip))
            {
                  if(fig instanceof NoteShape)
                  {
                    i = figures.indexOf(fig);
                    NoteShape.setName((String)TextObjectDiagram.elementAt(i));
                  }

                  if (fig instanceof GeneralizationLine){
                    i = figures.indexOf(fig);
                    GeneralizationLineClass Genl = (GeneralizationLineClass)TextObjectDiagram.elementAt(i);
                    GeneralizationLine.setName(Genl.getName());
                  }

                  if (fig instanceof AssociationLine){
                    i = figures.indexOf(fig);
                    AssociationLineClass Assl = (AssociationLineClass)TextObjectDiagram.elementAt(i);
                    AssociationLine.setName(Assl.getName());
                    AssociationLine.setAssocMultiA(Assl.getMultiplicityA());
                    AssociationLine.setAssocMultiB(Assl.getMultiplicityB());
                  }
```

```
            if (fig instanceof AgregationLine){
              i = figures.indexOf(fig);
              AggregationLineClass Agrl = (AggregationLineClass)TextObjectDiagram.elementAt(i);
              AgregationLine.setName(Agrl.getName());
              AgregationLine.setAggreMultiA(Agrl.getMultiplicityA());
              AgregationLine.setAggreMultiB(Agrl.getMultiplicityB());
            }

            if(fig instanceof ClassShape )
            {
              // Busca o Nome da classe seus atributos e metodos
              // para escrever na classe desenha na tela de edicao
             try{
              i = figures.indexOf(fig);
              SpecificationsClass SpecClas = (SpecificationsClass)TextObjectDiagram.elementAt(i);
              ClassShape.setName(SpecClas.getclassName());
              ClassShape.ClearVectors();
              if (SpecClas.allAttributes.size() > 0 || SpecClas.allMethods.size() > 0) {
                //Atributos
                for (Enumeration Attri = SpecClas.allAttributes.elements() ; Attri.hasMoreElements() ;){
                  AttributeClass AttCl = (AttributeClass)Attri.nextElement();
                  ClassShape.setallAtrib(AttCl.getName()+" : "+AttCl.getType());
                }
                //Metodos
                for (Enumeration Metho = SpecClas.allMethods.elements() ; Metho.hasMoreElements() ;)
                {
                  MethodClass MetCl = (MethodClass)Metho.nextElement();
                  ClassShape.setallMetho(MetCl.getName()+" : "+MetCl.getReturnType());
                }
              }
             }
             catch(Exception ex){System.out.println("Erro de conversao Classe.");}
            }
            fig.paint(g);
        }
        }
        g.setColor(handleColor());
        for (Enumeration e = handles.elements() ; e.hasMoreElements() ;) {
                Handle h = (Handle)e.nextElement();
                if (h.intersects(clip))
                        h.paint(g);
        }
}

protected void paintCompletely(Graphics g) {
        g.setColor(getForeground());
   int i=0;
        for (Enumeration e = figures.elements() ; e.hasMoreElements() ;) {
            if(figures.elementAt(i) instanceof NoteShape)
              NoteShape.setName((String)TextObjectDiagram.elementAt(i));
            if(figures.elementAt(i) instanceof ClassShape)
              ClassShape.setName((String)TextObjectDiagram.elementAt(i));
              i++;
           ((Figure)e.nextElement()).paint(g);
        }
        g.setColor(handleColor());
        for (Enumeration e = handles.elements() ; e.hasMoreElements() ;) {
           ((Handle)e.nextElement()).paint(g);
        }
}

public void repaint(Rectangle rectangle) {
        repaint(rectangle.x - 1, rectangle.y - 1, rectangle.width + 2, rectangle.height + 2);
}

public DrawingStyle style() {
        return style;
}
```

```
    public void setStyle(DrawingStyle style) {
            setForeground(style.foregroundColor());
            setBackground(style.backgroundColor());
            this.style = style;
    }

    public Figure[] figures() {
            Figure myFigures[] = new Figure[figures.size()];
            figures.copyInto(myFigures);
            return myFigures;
    }

    public Figure figureAt(int x, int y) {
            for (Enumeration e = new ReverseVectorEnumerator(figures); e.hasMoreElements();) {
                        Figure figure = (Figure)e.nextElement();
                        if (figure.inside(x,y)) return figure;
            }
            return null;
    }

    public Figure otherFigureAt(Figure excludedFigure, int x, int y) {
            for (Enumeration e = new ReverseVectorEnumerator(figures); e.hasMoreElements();) {
                        Figure figure = (Figure)e.nextElement();
                        if ((excludedFigure != figure) && figure.inside(x,y)) return figure;
            }
            return null;
    }

    public Handle handleAt(int x, int y) {
            for (Enumeration e = new ReverseVectorEnumerator(handles); e.hasMoreElements();) {
                        Handle handle = (Handle)e.nextElement();
                        if (handle.inside(x,y)) return handle;
            }
            return null;
    }

    public void addFigure(Figure figure) {
            figures.addElement(figure);
            figure.addObserver(this);
            repaint(figure.bounds());
       if(figure instanceof NoteShape) TextObjectDiagram.addElement(NoteShape.getNoteText().trim());
       if(figure instanceof LinkNoteLine) TextObjectDiagram.addElement("LinkNoteLine");

       if(figure instanceof AssociationLine) {
            AssociationLineClass AssocLine = new AssociationLineClass("","","","Element A: ","Element B: ","Unspecified
multiplicity","Unspecified multiplicity","");
            TextObjectDiagram.addElement(AssocLine);
       }
       if(figure instanceof AgregationLine) {
            AggregationLineClass AgreeLine = new AggregationLineClass("","","","Element A: ","Element B: ","Unspecified
multiplicity","Unspecified multiplicity","");
            TextObjectDiagram.addElement(AgreeLine);
       }

       if(figure instanceof GeneralizationLine) {
            GeneralizationLineClass General = new GeneralizationLineClass("",true,false,false,false,"",false,false);
            TextObjectDiagram.addElement(General);
       }

       if(figure instanceof ClassShape) {
            ClassDiagramEditor.ClassSpecific.Methods.removeAllElements();
            ClassDiagramEditor.ClassSpecific.Attributes.removeAllElements();
            SpecificationsClass SpecClas = new
SpecificationsClass("noname",0,0,ClassDiagramEditor.ClassSpecific.Attributes,ClassDiagramEditor.ClassSpecific.Methods, false);
            TextObjectDiagram.addElement(SpecClas);
            ClassDiagramEditor.ClassSpecific.dispose();
       }
    }

    public void addFigureBehind(Figure figure, Figure existingFigure) {
```

```
int existingIndex = figures.indexOf(existingFigure);
      if (existingIndex == -1) {
        figures.addElement(figure);
      if(figure instanceof AssociationLine) {
            AssociationLineClass AssocLine = new AssociationLineClass("","","","Element A: ","Element B: ","Unspecified
multiplicity","Unspecified multiplicity","");
            TextObjectDiagram.addElement(AssocLine);
      }

        if(figure instanceof LinkNoteLine) TextObjectDiagram.addElement("LinkNoteLine");
       if(figure instanceof AgregationLine) {
            AggregationLineClass AggreLine = new AggregationLineClass("","","","Element A: ","Element B: ","Unspecified
multiplicity","Unspecified multiplicity","");
            TextObjectDiagram.addElement(AggreLine);
      }

        if(figure instanceof GeneralizationLine) {
            GeneralizationLineClass General = new GeneralizationLineClass("",true,false,false,false,"",false,false);
            TextObjectDiagram.addElement(General);

        }
      }
      else {
         figures.insertElementAt(figure,existingIndex);
         if(figure instanceof AssociationLine) {
            AssociationLineClass AssocLine = new AssociationLineClass("","","","Element A: ","Element B: ","Unspecified
multiplicity","Unspecified multiplicity","");
            TextObjectDiagram.insertElementAt(AssocLine, existingIndex);
         }
         if(figure instanceof LinkNoteLine) TextObjectDiagram.insertElementAt("LinkNoteLine",existingIndex);

         if(figure instanceof AgregationLine) {
            AggregationLineClass AggreLine = new AggregationLineClass("","","","Element A: ","Element B: ","Unspecified
multiplicity","Unspecified multiplicity","");
            TextObjectDiagram.insertElementAt(AggreLine,existingIndex);
         }

      if(figure instanceof AssociationLine) {
            AssociationLineClass AssocLine = new AssociationLineClass("","","","Element A: ","Element B: ","Unspecified
multiplicity","Unspecified multiplicity","");
            TextObjectDiagram.insertElementAt(AssocLine,existingIndex);
      }

         if(figure instanceof GeneralizationLine) {
            GeneralizationLineClass General = new GeneralizationLineClass("",true,false,false,false,"",false,false);
            TextObjectDiagram.insertElementAt(General,existingIndex);
         }
      }
            figure.addObserver(this);
            repaint(figure.bounds());
   }

   public void moveFigureBehind(Figure figure, Figure existingFigure) {
            int index = figures.indexOf(figure);
            int existingIndex = figures.indexOf(existingFigure);
            if (index == -1 || existingIndex == -1) {
                        throw new IllegalArgumentException("At least one of the figures do not exist on this canvas");
            }
            if (index > existingIndex) {
                        figures.removeElement(figure);
            TextObjectDiagram.removeElementAt(index);
                        figures.insertElementAt(figure,existingIndex);

            if(figure instanceof AssociationLine) {
                  AssociationLineClass AssocLine = new AssociationLineClass("","","","Element A: ","Element B: ","Unspecified
multiplicity","Unspecified multiplicity","");
                  TextObjectDiagram.insertElementAt(AssocLine, existingIndex);
            }
            if(figure instanceof LinkNoteLine) TextObjectDiagram.insertElementAt("LinkNoteLine",existingIndex);
```

```
            if(figure instanceof AgregationLine) {
                AggregationLineClass AggreLine = new AggregationLineClass("","","","Element A: ","Element B: ","Unspecified
multiplicity","Unspecified multiplicity","");
                TextObjectDiagram.insertElementAt(AggreLine,existingIndex);
            }

            if(figure instanceof GeneralizationLine) {
                GeneralizationLineClass General = new GeneralizationLineClass("",true,false,false,false,"",false,false);
                TextObjectDiagram.insertElementAt(General,existingIndex);
            }
                    repaint(figure.bounds());
            }
    }

    public void moveFigureInFront(Figure figure, Figure existingFigure) {
            int index = figures.indexOf(figure);
            int existingIndex = figures.indexOf(existingFigure);
            if (index == -1 || existingIndex == -1) {
                    throw new IllegalArgumentException("At least one of the figures do not exist on this canvas");
            }
            if (index < existingIndex) {
                    figures.removeElement(figure);
                    figures.insertElementAt(figure,existingIndex);
            if(figure instanceof AssociationLine) {
                AssociationLineClass AssocLine = new AssociationLineClass("","","","Element A: ","Element B: ","Unspecified
multiplicity","Unspecified multiplicity","");
                TextObjectDiagram.insertElementAt(AssocLine, existingIndex);
            }
            if(figure instanceof LinkNoteLine) TextObjectDiagram.insertElementAt("LinkNoteLine",existingIndex);

            if(figure instanceof AgregationLine) {
                AggregationLineClass AggreLine = new AggregationLineClass("","","","Element A: ", "Element B: ","Unspecified
multiplicity","Unspecified multiplicity","");
                TextObjectDiagram.insertElementAt(AggreLine,existingIndex);
            }

            if(figure instanceof GeneralizationLine) {
                GeneralizationLineClass General = new GeneralizationLineClass("",true,false,false,false,"",false,false);
                TextObjectDiagram.insertElementAt(General,existingIndex);
            }
                    repaint(figure.bounds());
            }
    }

    public void moveFigureToBack(Figure figure) {
            if (figures.removeElement(figure))
       {
            int index = figures.indexOf(figure);

                    figures.insertElementAt(figure,0);
            TextObjectDiagram.insertElementAt(TextObjectDiagram.elementAt(i ndex),0);
       }
       else
                    throw new IllegalArgumentException("The figure does not exist on this canvas");
       repaint(figure.bounds());
    }

    public void moveSelectionsToBack() {
            for (Enumeration e = new ReverseVectorEnumerator(selections); e.hasMoreElements(); )
                    moveFigureToBack((Figure)e.nextElement());
    }

    public void moveSelectionsToFront() {
            for (Enumeration e = selections.elements(); e.hasMoreElements(); )
                    moveFigureToFront((Figure)e.nextElement());
    }

    public void moveFigureToFront(Figure figure) {
       if (figures.removeElement(figure)){
```

```
        int index = figures.indexOf(figure);
                figures.addElement(figure);
        TextObjectDiagram.addElement(TextObjectDiagram.elementAt(index));
  }
        else
                throw new IllegalArgumentException("The figure does not exist on this canvas");
  repaint(figure.bounds());
}

public void removeFigure(Figure figure) {
        figures.removeElement(figure);
        figure.deleteObserver(this);
        repaint(figure.bounds());
        figure.dispose();
}

public void removeDrawingLine(Figure figure) {
  int index = figures.indexOf(figure);
        figures.removeElement(figure);
        figure.deleteObserver(this);
  TextObjectDiagram.removeElementAt(index);
  repaint(figure.bounds());
        figure.dispose();
}

public Figure[] selections() {
        Figure mySelections[] = new Figure[selections.size()];
        selections.copyInto(mySelections);
        return mySelections;
}

public void addSelection(Figure figure) {
        if (!selections.contains(figure)) {
                selections.addElement(figure);
                addHandles(figure);
        }
}

public void toggleSelection(Figure figure) {
        if (selections.contains(figure)) removeSelection(figure);
        else addSelection(figure);
}

public void removeSelection(Figure figure) {
        removeHandles(figure);
        selections.removeElement(figure);
        if (figure.isObsolete())
                removeFigure(figure);
}

public void select(Figure figure) {
        clearSelections();
        addSelection(figure);
}

public void clearSelections() {
        for (Enumeration e = selections.elements(); e.hasMoreElements(); ) {
                Figure figure = (Figure)e.nextElement();
                if (figure.isObsolete())
                        removeFigure(figure);
        }
        selections.removeAllElements();
        clearHandles();
}

public Handle[] handles() {
        Handle myHandles[] = new Handle[handles.size()];
        handles.copyInto(myHandles);
        return myHandles;
}
```

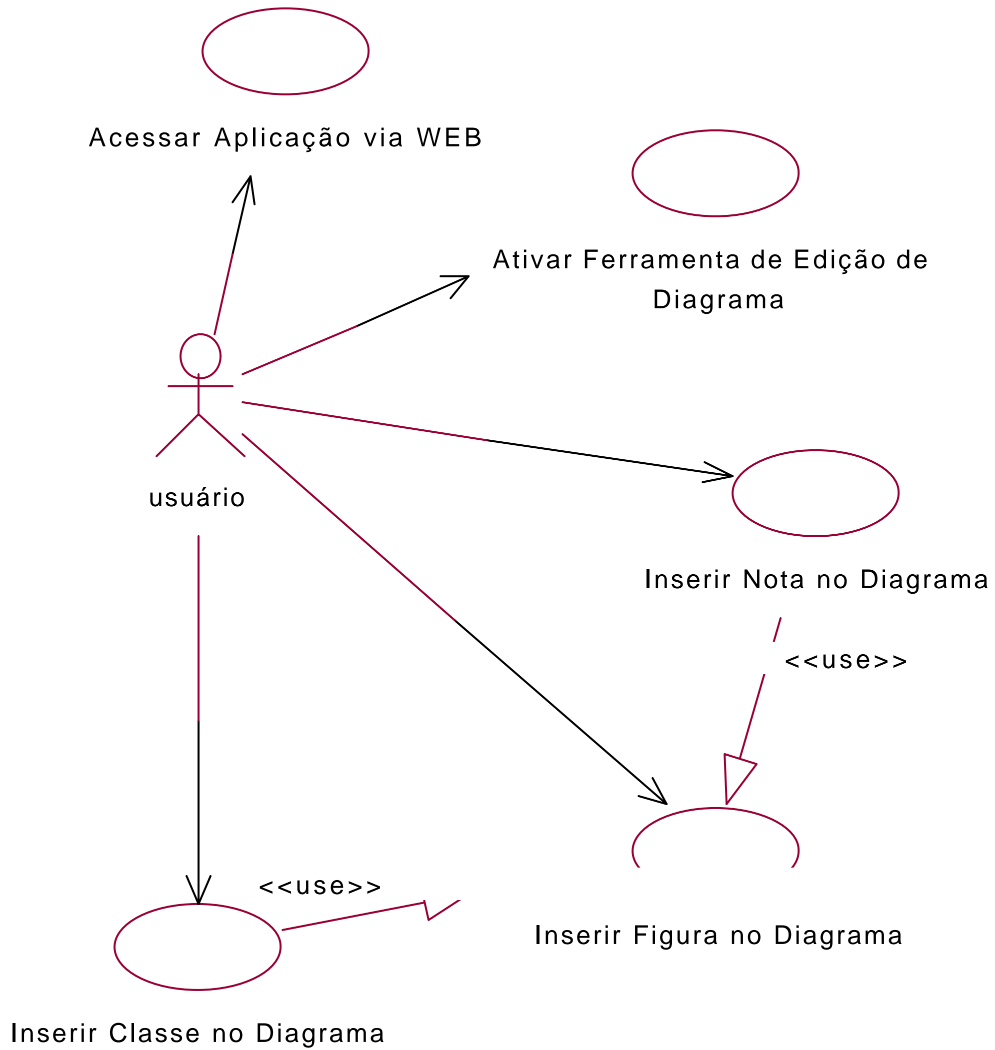
```
public void addHandles(Figure figure) {
        Handle handles[] = figure.handles();
        figureHandles.put(figure, handles);
        addHandles(handles);
}

public void addHandles(Figure figure, Handle handles[]) {
        Handle oldHandles[] = (Handle[])figureHandles.get(figure);
        if (oldHandles == null)
                figureHandles.put(figure, handles);
        else {
                Handle newHandles[] = new Handle[oldHandles .length + handles.length];
                System.arraycopy(oldHandles,0,newHandles,0,oldHandles.length);
                System.arraycopy(handles,0,newHandles,oldHandles.length,handles.length);
                figureHandles.put(figure, newHandles);
        }
        addHandles(handles);
}

public void removeHandles(Figure figure) {
        Object handles = figureHandles.remove(figure);
        if (handles != null)
                removeHandles((Handle[])handles);
}

public void addHandle(Handle handle) {
        this.handles.addElement(handle);
        repaint(handle.bounds());
}

public void removeHandle(Handle handle) {
        if (this.handles.removeElement(handle))
                repaint(handle.bounds());
}

public boolean handleEvent(Event evt) {
        if (this.tool().handleEvent(evt)) return true;
        return super.handleEvent(evt);
}

public EventHandler tool() {
   if (tool == null) return defaultTool();
        return tool;
}

public void setTool(EventHandler newTool, int IndexButton)
{
        if (tool instanceof Handle)
                ((Handle)tool).releaseControl(this);
        tool = newTool;
   if (tool==null && selections.size() == 1) {
      if (IndexButton == 8 && selections.elementAt(0) instanceof ClassShape)
      {
         ClassDiagramEditor.SuggestionButton_Action();
      }
   }
}

public void toolTaskCompleted(EventHandler tool) {
}

protected EventHandler defaultTool() {
        return new SelectionTool(this);
}

protected DrawingStyle defaultStyle() {
        return new SimpleDrawingStyle();
}
```

```
protected void addHandles(Handle handles[] ) {
        for (int i=0; i < handles.length; i++)
                this.handles.addElement(handles[i]);
        repaint();
}

protected void removeHandles(Handle handles[]) {
        for (int i=0; i < handles.length; i++)
                this.handles.removeElement(handles[i]);
        repaint();
}

protected void clearHandles() {
        handles.removeAllElements();
        figureHandles.clear();
        repaint();
}

public void update(Figure figure, Object arg)  {
        Rectangle area = figure.bounds();
        if (arg instanceof Rectangle) {
                area = area.union((Rectangle)arg);
        } else if (arg instanceof Point) {
                Point point = (Point)arg;
                area = new Rectangle(area.x, area.y, area.width, area.height);
                area.add(point);
                area.add(point.x + area.width, point.y + area.height);
        } else if (arg instanceof Dimension) {
                Dimension oldSize = (Dimension)arg;
                if (oldSize.width > area.width)
                        area = new Rectangle(area.x, area.y, oldSize.width, area.height);
                if (oldSize.height > area.height)
                        area = new Rectangle(area.x, area.y, area.width, oldSize.height);
        }
        repaint(area);
}

public void update(Observable subject, Object arg)  {
        System.err.println("update with Object not Figure");
}

public void deleteSelections()  {
        Vector oldSelections = (Vector)selections.clone();

   for (Enumeration s = selections.elements(); s.hasMoreElements();) {
        int index = figures.indexOf((Figure)s.nextElement());
        TextObjectDiagram.removeElementAt(index);
   }
        for (Enumeration e = oldSelections.elements(); e.hasMoreElements();) {
                removeFigure((Figure)e.nextElement());
        }
        clearSelections();
}

public void copySelections()  {
        putToClipboard(duplicateFigures(validateOrder(selections)));
   SelectedFigures.removeAllElements();
   SelectedText.removeAllElements();
   putToSelectedFigures(validateOrder(selections));
}

public void paste()  {
   clearSelections();
        Vector duplicates = duplicateFigures(getFromClipboard());
        for (Enumeration e = duplicates.elements(); e.hasMoreElements();) {
                Figure duplicate = (Figure)e.nextElement();
                duplicate.translate(5,5);
        figures.addElement(duplicate);
        duplicate.addObserver(this);
        repaint(duplicate.bounds());
```

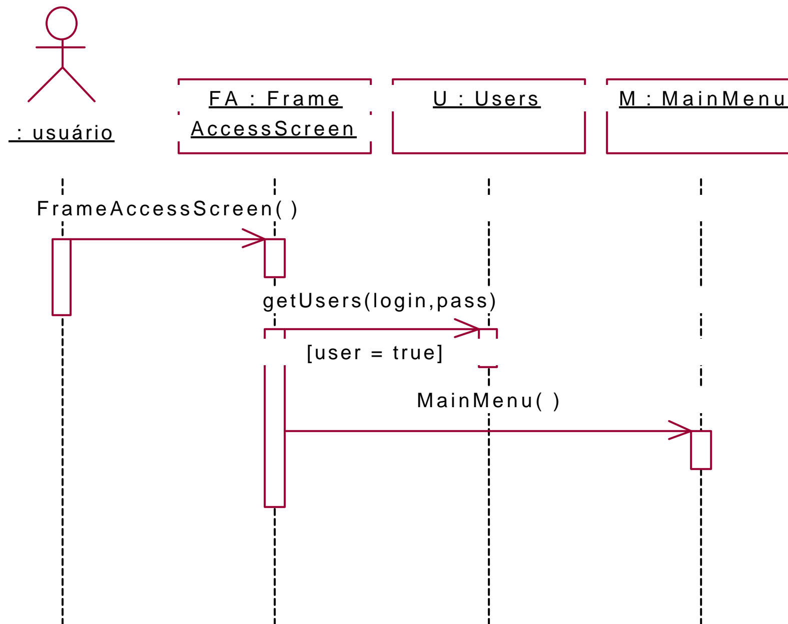
```
                    addSelection(duplicate);
          }
     for (int i = 0; i < SelectedFigures.size(); i++) {
          Figure duplicate = (Figure)SelectedFigures.elementAt(i);
          if(duplicate instanceof NoteShape) TextObjectDiagram.addElement((String)SelectedText.elementAt(i));
          if(duplicate instanceof AssociationLine) {
               AssociationLineClass Al = (AssociationLineClass)TextObjectDiagram.elementAt(i);
               AssociationLineClass Assl = new AssociationLineClass(Al.getName(), Al.getRoleA(), Al.getRoleB(), Al.getElementA(),
Al.getElementB(), Al.getMultiplicityA(), Al.getMultiplicityB(),Al.getDocumentation());
               TextObjectDiagram.addElement(Assl);
          }
          if(duplicate instanceof LinkNoteLine) TextObjectDiagram.addElement("LinkNoteLine");

          if(duplicate instanceof AgregationLine) {
               AggregationLineClass Ag = (AggregationLineClass)TextObjectDiagram.elementAt(i);
               AggregationLineClass Agrl = new AggregationLineClass(Ag.getName(), Ag.getRoleA(), Ag.getRoleB(), Ag.getElementA(),
Ag.getElementB(), Ag.getMultiplicityA(), Ag.getMultiplicityB(),Ag.getDocumentation());
               TextObjectDiagram.addElement(Agrl);
          }

          if(duplicate instanceof GeneralizationLine) {
               GeneralizationLineClass Gl = (GeneralizationLineClass)TextObjectDiagram.elementAt(i);
               GeneralizationLineClass Genl = new
GeneralizationLineClass(Gl.getName(),Gl.getPublic(),Gl.getProtected(),Gl.getPrivate(),Gl.getImplementation(),Gl.getDocumentation(),Gl.getFrie
ndShip(),Gl.getVirtual());
               TextObjectDiagram.addElement(Genl);
          }
          if (duplicate instanceof ClassShape){
            SpecificationsClass SC = (SpecificationsClass)SelectedText.elementAt(i);
            SpecificationsClass SpecClas = new
SpecificationsClass(SC.getclassName(),SC.getattributeNumber(),SC.getmethodNumber(),SC.getallAttributes(),SC.getallMethods(),SC.getSugges
tions());
            TextObjectDiagram.addElement(SpecClas);
          }
     }
   }

   public void putToSelectedFigures(Vector stuff)  {
   //Copia de figuras para posterior colagem
      for (Enumeration e = stuff.elements(); e.hasMoreElements();)
          SelectedFigures.addElement(e.nextElement());

   //Copia dos textos das figuras para posterior colagem
      for (Enumeration e = SelectedFigures.elements(); e.hasMoreElements();) {
                    Figure duplicate = (Figure)e.nextElement();
          int index = figures.indexOf(duplicate);
          if(duplicate instanceof NoteShape) SelectedText.addElement((String)TextObjectDiagram.elementAt(index));
          if(duplicate instanceof AssociationLine) {
               AssociationLineClass Al = (AssociationLineClass)TextObjectDiagram.elementAt(index);
               AssociationLineClass Assl = new AssociationLineClass(Al.getName(), Al.getRoleA(), Al.getRoleB(), Al.getElementA(),
Al.getElementB(), Al.getMultiplicityA(), Al.getMultiplicityB(),Al.getDocumentation());
               SelectedText.addElement(Assl);
          }
          if(duplicate instanceof LinkNoteLine) SelectedText.addElement("LinkNoteLine");
          if(duplicate instanceof AgregationLine) {
               AggregationLineClass Ag = (AggregationLineClass)TextObjectDiagram.elementAt(index);
               AggregationLineClass Agrl = new AggregationLineClass(Ag.getName(), Ag.getRoleA(), Ag.getRoleB(), Ag.getElementA(),
Ag.getElementB(), Ag.getMultiplicityA(), Ag.getMultiplicityB(),Ag.getDocumentation());
               SelectedText.addElement(Agrl);
          }
          if(duplicate instanceof GeneralizationLine) {
               GeneralizationLineClass Gl = (GeneralizationLineClass)TextObjectDiagram.elementAt(index);
               GeneralizationLineClass Genl = new
GeneralizationLineClass(Gl.getName(),Gl.getPublic(),Gl.getProtected(),Gl.getPrivate(),Gl.getImplementation(),Gl.getDocumentation(),Gl.getFrie
ndShip(),Gl.getVirtual());
               SelectedText.addElement(Genl);
          }
          if (duplicate instanceof ClassShape){
            SpecificationsClass SC = (SpecificationsClass)TextObjectDiagram.elementAt(index);
```

```
                SpecificationsClass SpecClas = new
SpecificationsClass(SC.getclassName(),SC.getattributeNumber(),SC.getmethodNumber(),SC.getallAttributes(),SC.getallMethods(),SC.getSugges
tions());
                SelectedText.addElement(SpecClas);
            }
        }
}
    public void cutSelections()  {
            copySelections();
            deleteSelections();
    }

    public void putToClipboard(Vector stuff)  {
            clipboard = stuff;
    }

    public Vector getFromClipboard()  {
            return clipboard;
    }

    protected Vector duplicateFigures(Vector toCopy)  {
            Vector copy = new Vector(figures.size());
            Hashtable duplicates = new Hashtable();

            for (Enumeration e = toCopy.elements( ); e.hasMoreElements();) {
                    Figure original = (Figure)e.nextElement();
                    Figure duplicate = (Figure)original.duplicate();
                    duplicates.put(original,duplicate);
                    copy.addElement(duplicate);
            }
            for (Enumeration e = copy.elements(); e.hasMoreElements();) {
                    Figure duplicate = (Figure)e.nextElement();
                    duplicate.postDuplicate(duplicates);
            }
            return copy;
    }
    protected Vector validateOrder(Vector unordered)  {
            Vector copy = new Vector(unordered.size());
            int positions[] = new int[unordered.size()];
            int i = 0;
            for (Enumeration e = unordered.elements(); e. hasMoreElements(); i++) {
                    Object element = e.nextElement();
                    positions[i] = figures.indexOf(element);
                    int count = 0;
                    for (int j=0; j < i; j++)
                            if (positions[i] > positions[j]) count++;
                    copy.insertElementAt(element,count);
            }
            return copy;
    }
    protected Color handleColor() {
            return style.foregroundColor();
    }
}
```

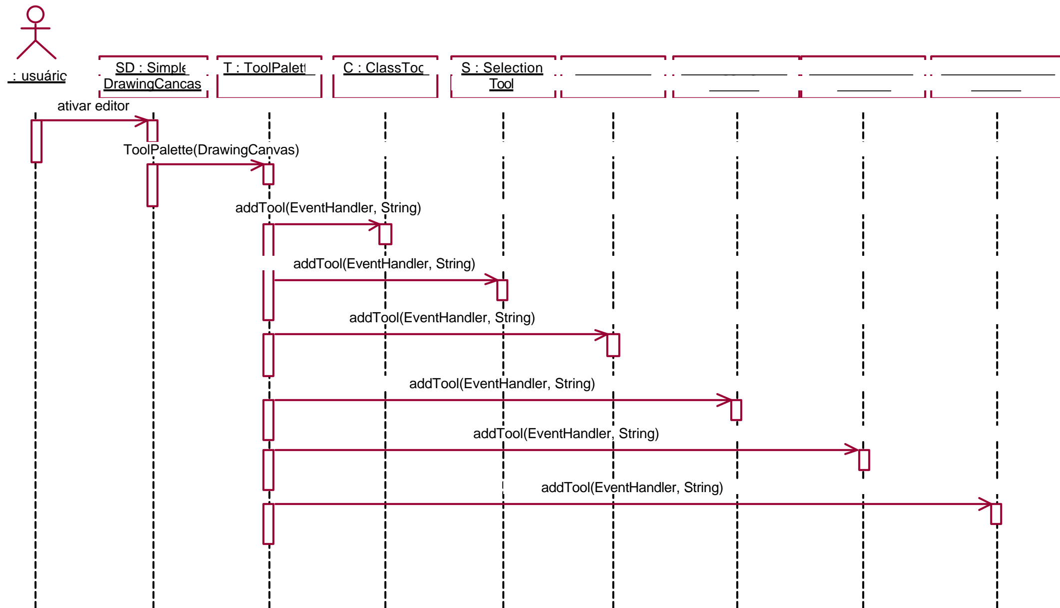
Anexo C Diagrama de caso de uso: Acesso ao editor via *Web*



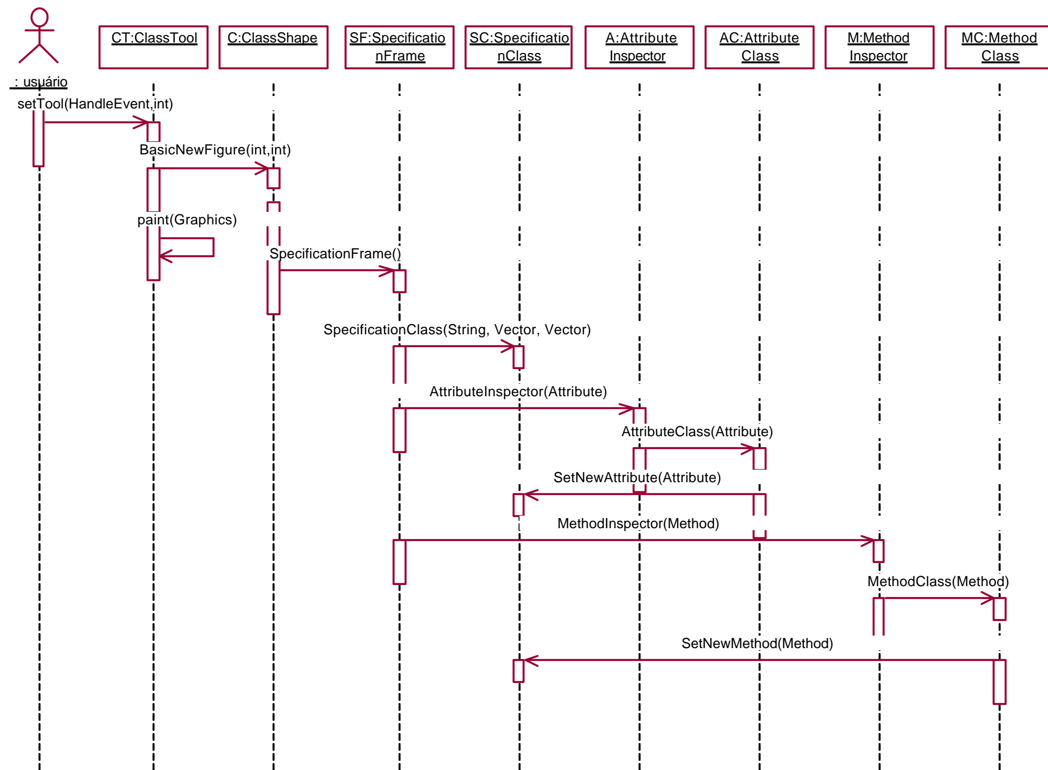
Anexo D Diagrama de Seqüência: Acesso ao editor via *Web*



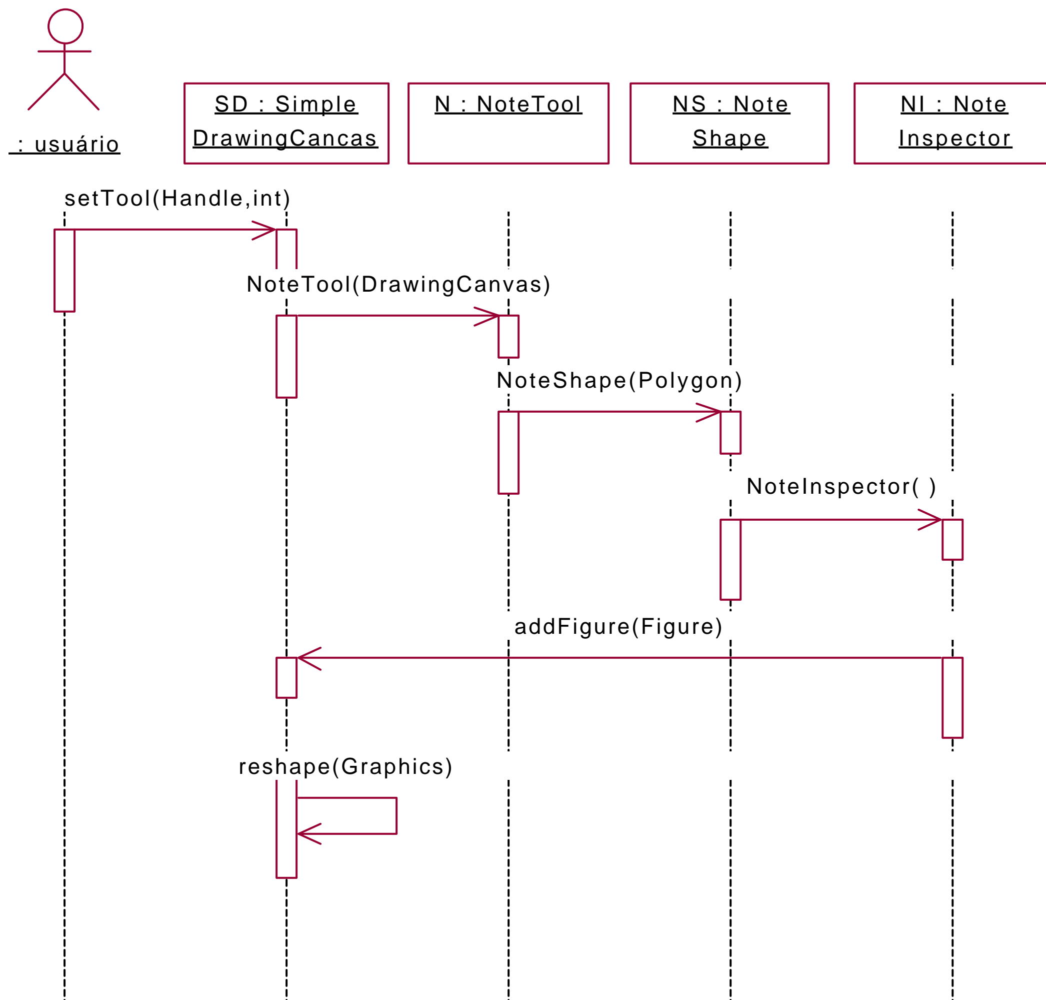
Anexo E Diagrama de Seqüência: Ativar ferramenta de edição de diagramas



Anexo F Diagrama de Seqüência: Inserir classe no diagrama



Anexo G Diagrama de Seqüência: Inserir nota no diagrama



Bibliografia

[ALM 2000] ALMEIDA, Islenho de; FOLTRAN JÚNIOR, Dierone César. Tecnologia Java aplicada no desenvolvimento de aplicativos Web distribuídos. In: SIMPÓSIO CATARINENSE DE COMPUTAÇÃO, 1., 2000, Itajaí. **Anais...** Itajaí: UNIVALI, 2000. p. 551 – 561.

[ARB 95] ARBORTEXT INC. **A Guide to SGML (Standard Generalized Markup Language) and Its Role in Information Management**. Disponível em: <http://www.arbotext.com/Think_Tank/SGML_Resources/Getting_Started_with_SGML/getting_started_with_sgml.html>. Acesso em: 13 set. 2000.

[BAH 86] BAHLKE, R.; SNELTING, G. The SPG System: From Formal Lanuage Definitons to Interactive Programming Environments. **ACM Transactions on Programming Languages and System**, New York, v.8, n.4, p.547-76, Oct. 1986.

[BAR 94] BARROS, Ligia A. **Suporte a Ambientes Distribuidos para Aprendizagem Cooperativa.** Rio de Janeiro: COPPE/UFRJ, 1994. Tese de Doutorado.

[BAR 94] BARROS, L.A.. Suporte a Ambientes Distribuídos para Aprendizagem Cooperativa . Tese de doutorado - COPPE/UFRJ, 1994.

[BOO 91] BOOCH, G. **Object Oriented Design**: with Applications. Redwood City: The Benjamin/Cumming, 1991.

[BOO 99] BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **The Unified Modeling Language user guide**. USA: Addison-Wesley, 1999.

[BOR 95] BORGES, Marcos R; CAVALCANTI, Maria C; CAMPOS, Maria L. **Suporte po Computador ao Trabalho Cooperativo**, XIV Jornada de Atualização em Informática, Canela-RS, 1995.

[BRA 95] BRANT, J. **HotDraw**. Urbana: University of Ilinois as Urbana-champaign, 1995. Master thesis.

[BRA 95a] BRANT, John Machael. **HotDraw**. Urbana-Champaing: University of Illinois, 1995. 48 p. (Dissertação de Mestrado). Disponível em: <http://st-www.cs.uiuc.edu/ftp/pub/papers/HotDraw.ps.gz>. Acesso em: 15 ago. 1998.

[BRA 95b] BRANT, John Machael; JOHNSON, Ralph E. **Creating Tools in HotDraw by Composition**. Disponível em: <http://st-www.cs.uiuc.edu/ftp/pub/papers/HotDrawTools.ps.gz>. Acesso em: 15 ago. 1998.

[BRO 2000] BRONFMAN, Paula Nemetz. **Estudo de Padrões de Documentos XML para o Comércio Eletrônico Business to Business**. Porto Alegre: CPGCC da UFRGS, 2000.

[BRU 97] BRUGALI, Davide; MENGA, Guiseppe; AARSTEN, Amund. The Framework Life Span. **Communications of the ACM**, New York, v.40, n.10, p.65-68, Oct. 1997.

[BRY 97] BRYAN, Martin. **An Introduction to the Extensible Markup Language (XML)**. Disponível em: <http://www.personal.u-net.com/~sgml/xmlintro.html>. Acesso em: 29 out. 2000.

[BUS 96] BUSCHMANN, Frank; MEUNIER, Regine; ROHNERT, Hans; SOMMERLAND, Peter; STAL, Michael. **Pattern-Oriented Software Architecture**: System of Patterns. John Wiley & Sons Ltd: England, 1996.

[CAM 97] CAMPO, M. **Compreensão Visual de Frameworks através de Introspecção de Exemplos**. Porto Alegre: CPGCC da UFRGS, 1997.

[CAS 99] CASTRO, Luciana B. **Componente Gerador de Documentos XML e Ferramenta para Criação e Visualização de Relatórios**. 1999.

[CHA 98] CHAN, Mark C. et al. **Java – 1001 dicas de programação**. São Paulo: Makron books, 1998.

[COL 2000] COLCHER, Raul. **Guia de EDI e Comércio Eletrôcino**. Rio de Janeiro: Simpósio-Brasil, 1999-2000. 171 p.

[COL 98] COLLA, E. C. Arquitetura dos Servilets: Java do Lado do Servidor. **Developer's Magazine** , São Paulo, v.23, n.7, p. 16-18. July 1998.

[CON 2000] CONALLEN, Jim. ***Building Web Applications with UML***. Reading: Addison Wesley, 2000.

[CON 97] CONNOLLY Dan; KHARE Rohit; RIFKIN Adam. The Evolution of Web Documents: The Ascent of XML. **World Wide Web Journal Special Issue on XML**, v.2, n.4, p119-128, 1997.

[CRU 98] CRUZ, Tadeu. **Workflow:** a tecnologia que vai revolucionar processos. São Paulo: Atlas, 1998.

[DAV 98] DAVIDSON, James Duncan; AHMED, Suzanne. **Java Servlet API Specification** Version 2.1a. Sun Microsystems Inc, 1998.

[DEU 89] DEUTSCH, P. *Frameworks* and reuse in the smalltolak 80 system. In: BIGGERSTAFF, Ted. **Software reusability**: Aplication and Experience. New York: ACM Press, 1989. v.1, p. 57-71.

[DIE 96] DIETRICH, Elton. **Projeto de um Sistema de Suporte à Autoria Cooperativa de Hiperdocumentos**. Porto Alegre: CPGCC da UFRGS, 1996, 125p. Dissertação de Mestrado.

[DIL 94] DILLENGOURG,P. et al. 1994. The evolution of research on collaborative learning. Disponível por WWW em http://tecfa.unice.ch/ tecfa/research/lhm/ESF-Chap5.text.

[DIS 2000] DISTEC, EDI. **What is EDI?**. Disponível em: <http://www.interlog.com/~cwhiting/qhatis.html>. Acesso em: 15 set. 2000.

[DON 84] DONZEAU-GOUGE, V. et al. Document Structure and Modularity in MENTRO. **SINGPLAN Notices**, New York, v.19, n.5, p.141-148, May 1984.

[EDW 97] EDWARDS. Jeri. **3-tier Client/Server at Work**. John Wiley Computer, 1997.

[ERI 98] ERIKSSON, H.; PENKER, M. **UML ToolKit**. USA: John Wiley & Sons, 1998.

[ESP 93] ESPERANÇA, L. G; PRICE, R. T. **A Implementação de um Gerador de Editores Dirigidos por Sintaxe e seu uso para uma Linguagem de Especificação Formal**. Porto Alegre: CPGCC da UFRGS. 1993.

[FAR 98] FARLEY, Jim. **JAVA™** ***Distributed Computing***. Sebastopol: O'Reilly, 1998.

[FAV 88] FAVERO, E. L.; PRICE, R. T. A implementação de Editores Dirigidos por Sintaxe: Algumas Considerações. In: REUNIÃO DE TRABALHO E COLETÂNEA DE RESULTADOS DE PESQUISA, 3. 1988. **Projeto Estrela**. São Paulo: Sid-Informática, 1988. p.269-280.

[GAM 95] GAMMA, E. et al. **Design Patterns:** *elements of reusable object-oriented software*. Reading: Addsion Wesley, 1995.

[GOC 98] GOLDFRAK, Charles F.; PRESCOD, Paul. **The XML Handbook**. Prentice Hall. 1998.

[GOL 95] GOLDBERG, A.; RUBIN, K. **Succeeding with Objects**: Decision Frameworks for Projet Management. Reading: Addison-Wesly, 1995.

[GRU 94] GRUDIN, J. Computer-Supported Cooperative Work: **History and Focus**. Computer, mai., 1994.

[HAB 86] HABERMANN, A. N., NOTKIN, D. Candalf: Software Development Environments. **IEEE Transactions on Software Engineering**, New York, v.12, n.12, p.1117-1127, Dec. 1986.

[HAM 99] HAMPSHIRE, P. P. Utilizando Java como Ferramenta Corporativa. **Developers' Magazine** , São Paulo, v.34, n.6, p.36-39. June 1999.

[HAN 71] HANSEN, W. **Creation of Hierarchic Text With a Computer Desplay**. Stanford: Stanford University, Departmente of Computer Science, 1971.

[JOH 92] JOHNSON. R. Documenting Frameworks Using Patterns. In: **Conference on Object-Oriented Programming, Languages and Applications**, 8., 1992, Washington DC. **Tutorial Notes...** [S.l:s.n.], 1992.

[JOH 97] JOHNSON, R. E. **Components, frameworks, patterns**. Feb. 1997. Disponível por FTP anônimo em st.cs.uiuc.edu (dez. 98).

[KHO 95] KHOSHAFIAN, Setrag, BUCKIEWICZ, Marek. **Introduction to groupware, workflow and workgroup computing**. New York: John Wiley, 1995.

[KNO 97] KNOWLEDGE SYSTEMS CORPORATION. **Java Version of HotDraw**. Disponível em <http://www.ksccary.com.ksc/HotDraw.HotDraw>. Acesso em: 20 jan. 2000.

[KRO 99] KROENKE, David M. **Banco de dados:** Fundamentos, Projeto e Implementação. 6. ed. Rio de Janeiro: LTC, 1999.

[LAR 99] LARMAN, Craig. **Applying UML and Patterns** – An Introduction to Object-Oriented Analysis and Design. USA: Perntice-Hall, 1999.

[MAC 89] MACHADO, Javan de Castro, PRICE, R. T. Modelagem dos Dados de um Ambiente de Desenvolvimento de *Software*. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 3., 1989, Recife. **Anais...** Recife: SBC/UFPE, 1989. p. 296-310.

[MAC 97] MACHERINS, Ingo. **XML : a professional alternative to HTML:** Expert's Revolution. Disponível em: <http://www.hese.de/ix/artikel/E/1997/06/106/artikel.html>. Acesso em: 13 jul. 1998.

[MAR 89a] MARTIN, James; MACCLURE, C. **Digramming Techniques for analysts and programmers**. Englewood Cliffes: Prentice-Hall, 1985.

[MAR 89b] MARTIN, James; MACCLURE, C. **Techniques Computing**. Englewood Cliffes. Prentice-Hall, 1985.

[MEL 89] MELO, W. L. M.; PRICE, R. T. **Proposta de um Editor Diagramático Generalizado**. Porto Alegre: PGCC da UFRGS, 1989.

[NEU 90] NEURWITH, C. M. et al. Issues in the Design of Computer Support for Co-authoring and Commenting. In: **Proceedings fo Computer Supported Cooperative Work**. 1990.

[NOT 2000] NOTARI, Daniel Luís. **Uma enciclopédia para ambientes distribuídos de desenvolvimento de software**. Porto Alegre: PPGC da UFRGS, 2000.

[OTS 97] OTSUKA, J. L. Proposta de um Sistema de Aprendizagem Colaborativa baseado no WWW. In: SIMPÓSIO BRASILEIRO DE INFORMÁTICA NA EDUCAÇÃO, 8., 1997, São José dos Campus, SP. Anais ... São José dos Campus: SBC, 1997.

[PAP 91] PAPPAS, Chris H., MURRAY, Willian H. **Turbo c++ completo e total**. trad. Mário Moro Fecchio; revisão técnicas José Eduardo Maluf de Carvalho. São Paulo: Makron, MacGraw-Hill, 1991.

[PAR 94] PARCPLACE SYSTEMS. **VisualWorks User's Guide**. Sunnyvale: ParcPlace Systems, 1994.

[PEN 93] PENEDO, M. Process-based Software Engineering Environments (PSEE). In: BRAZILIAN SYMPOSIUM OF SOFTWARE ENGINEERING, 7., 1993, Rio de Janeiro. **Anais...** Rio de Janeiro: PUC do Rio de Janeiro, 1993.

[POM 99] POMPOERMAIER, L. B.; PRICE, R. T. Um Editor Diagramático para Desenvolvimento de Sistemas de Informação na Internet. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 13., 1998, Maringá. **Anais...** Maringá: SBES/UFPR, 1998. p. 265-277.

[PRE 94] PREE, W. Meta-Patterns: Abstracting the Essentials of Object-Oriented Frameworks. In: european conference on object-oriented programming, 10., 1994, Bolonga, Italia. **Proceedings...** Berlin: Springer-Verlag, 1994. p.150-164.

[PRE 95] PREE, Wolfang. **Design Patterns for Object – Oriented Software Development**. Reading: Addison-Wesley, 1995.

[RAT 96] Rational Software Corp., "Unified Modeling Language". Disponível em: <http://www.rational.com/>. Acesso em : 20 out. 1999.

[REE 92] REEVES B.; SHIPMAN, F. Supporting Communication between Designers with Artifact-Centered Evolving Information Spaces. In: **Proceedings of Conferece on Computer Supported Collaborative Work**. November 1992. p. 394-401.

[REI 99] REIS, G. S. RMI – Remote Method Invocation. **Developers' Magazine**. São Paulo, v.33, n.6, p. 52-54, maio 1999.

[REP 88] REPS, T., TEITELDAUM, T. **The Synthesizer Generator:** A system for Contructing Language-Based Editors. New York: Springer-Verlag, 1988. p.317.

[RMI 2000] Getting Started Using RMI. Disponível em: <http://www.java.sun.com/j2se/1.3/docs/guide/rmi/ getstart.doc.html>. Acesso em: 20 jun. 2000.

[RMI 97] RMI Documentation. Disponível em: <http://ww.java.sun.com/products/jdk/1.1/docs/guide/rmi/index.html>. Acesso em: 15 out. 1997.

[RUM 91] RUMBAUGH, J. et al. **Object-Oriented Modelling and Design.** Englewood Cliffs: Prentice Hall. 1991.

[SAP 2001] SAPSOMBOON, B., ANDRIATI, R., ROBERTS, L., SPRING, M.B. Software to aid collaboration: focus on collaborative authoring. Disponível em http://www.sis.pitt.edu/~spring/cas/cas.html. Acesso em: 06 abr. 2001.

[SCH 92] SCHMIDT, K. and BANNON, L. J. Taking CSCW Seriously - Supporting Articulation Work. Computer Supported Cooperative Work, 1(1-2): 7-40. 1992.

[SIL 2000] SILVA, Ricardo P. **Suporte ao desenvolvimento e uso de frameworks e componentes**. Porto Alegre: PPGC da UFRGS, 2000.

[SIL 96] SILVA, Ricardo P. **Avaliação de metodologias de análise e projeto orientados a objetos voltados ao desenvolvimento de aplicações, sob a ótica de sua utilização no desenvolvimento de *frameworks* orientados a objetos**. Porto Alegre: UFRGS/II/CPGCC, Julho de 1996. (TI - 556).

[SIM 89] SILVA, Mônica Spotorno da. **Um Formatador de Diagramas**. Porto Alegre: PGCC da UFRGS, 1989. Dissertação de Mestrado.

[SOU 97] SOUZA, C. R. B.; WAINWE, J.; RUBIRA, C. M. F. Um modelo de Anotações para o Desenvolvimento Cooperativo de Software. In: Workshop on Hypermedia e Multimedia Applications, 3., 1997, São Carlos. **Anais...** São Carlos, 1997. P. 143-154.

[THE 87] THE SCOPE of Workbench Products. In: **Analyst Workbenches**. Oxford: Pergaman Infotech, 1987. p. 199-313.

[UNA 91] UNAMAKER, J.F. "Electronic meeting systems to support group work", Communications of the ACM, 34 (7), Jul 1991, pp. 40-61.

[VIN 98] VINOSKI, S. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. Disponível em: <http://www.iona.com/hiplan/vinoshi/ieee.pdf>. Acesso em: mar. 1998.

[W3C 98] W3C. **Extensible Markup Language (XML) 1.0**. Disponível em: <http://www.w3.org.TR/1998/REC-xml-19980210>. Acesso em: 13 jun. 2000.

[WEG 96] WEGNER, Peter. Interactive Software Technology. In: TUCKER, A. (Ed.). **Handook of Computer Science an Engeneering**. Boca Raton: CRC Press, 1996. Disponível em: <http://www.cs.brown.edu/people/pw>. Acesso em: jul. 1999.

[ZHO 99] ZHONGHUA, Y. & DUDDY, K. **CORBA**: A Platform for Distributed Object Computing. Disponível em: <http://www.omg.org>. Acesso em: jul. 1999.