UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

PROTÓTIPO DE FERRAMENTA CASE PARA GERAÇÃO DE CÓDIGO C++ E DIAGRAMA DE CLASSES

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

VANDERLEI BALLMANN

BLUMENAU, JUNHO/2000

2000/1-67

PROTÓTIPO DE FERRAMENTA CASE PARA GERAÇÃO DE CÓDIGO C++ E DIAGRAMA DE CLASSES.

VANDERLEI BALLMANN

ESTE TRABALHO DE CONCLUSÃO DE CURSO, FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. Wilson Pedro Carli — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. Wilson Pedro Carli

_____

Prof. Marcel Hugo

_____

Prof. Roberto Heinzle

DEDICATÓRIA

A meus pais, irmãos e namorada,
sem eles esse trabalho não se concretizaria.

AGRADECIMENTOS

Ao meu orientador, Professor Wilson Pedro Carli. Sua orientação neste trabalho foi muito importante para a conclusão deste TCC.

Agradeço, principalmente, ao meu pai, Aloisio Ballmann (falecido) e minha mãe, Valéria Steiner Ballmann, que mesmo na ausência do meu pai, sempre lutou muito e tornou a minha caminhada muito mais suave. Agradeço também aos meus irmãos, Dolores, Tânia e Jurandir, pelo carinho demonstrado.

Agradeço a minha namorada Karina pela ajuda e compreensão nas horas difíceis.

Agradeço, também a todos da Senior Sistemas, em especial a Guido Heinzen, pelos conselhos e orientação na minha vida profissional. Com certeza a Senior é um lugar muito especial para se trabalhar.

Sumário

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS

ANSI – *American National Standards Institute*.

CASE – *Computer Aided Software Engineering*, ou Engenharia de Software Auxiliada por computador.

CASE-OO – Ferramentas CASE orientadas a objeto.

I-CASE – *Integrated CASE*, Ferramentas CASE Integradas.

MDI – *Multiple Document Interface*, interface com múltiplos documentos.

OMT – *Object Modeling Technique*, ou Técnica de Modelagem de Objetos.

OO – Orientação a Objetos.

OOSE – *Object-Oriented Software Engineering*.

UML – Unified Modeling *Language*, Linguagem de Modelagem Unificada.

RESUMO

Este trabalho apresenta um estudo sobre a linguagem de modelagem de objetos unificada (UML - *Unified Modeling Language)*, visando desenvolver uma ferramenta CASE para geração de código fonte em C++ a partir da definição de um diagrama da ferramenta Rational Rose. Além disso, a ferramenta é capaz de realizar a engenharia reversa em nível de código fonte para gerar o diagrama de classes do Rational Rose a partir de um fonte em C++.

ABSTRACT

This work shows a study about the Unified Modeling Language (UML) that is aimed to the development of a CASE tool to generate C++ source code from a Rational Rose diagram definition. Beyond this, this tool is able to fullfil the reverse engineering at source code level to generate the Rational Rose class diagram from a C++ source code.

1 INTRODUÇÃO

A competitividade, a necessidade de desenvolvimento mais rápido e de softwares mais confiáveis têm levado as empresas a desenvolverem e utilizarem novas e melhores tecnologias de modelagem e desenvolvimento de sistemas. A indústria da informática vem oferecendo soluções que buscam minimizar dificuldades através de ferramentas de modelagem que simulam modelos da realidade de forma mais amigável através de ambientes gráficos e interfaces ricas ([FUR1998]).

Criar aplicações a partir de componentes não é algo novo, esse enfoque tem estado em uso durante vários anos. Assim, pergunta-se : por que 100% das organizações ainda não adotaram a tecnologia de objetos? Uma forma final de resistência está no fato que muitas organizações têm um investimento significativo em ferramentas não orientadas a objeto e treinamento de seus técnicos ([FUR1998]).

Vários métodos orientados a objeto (por exemplo, Rumbaugh, OOSE, Booch, OMT, Coad/Yourdon, etc) foram desenvolvidos com o objetivo de melhorar a modelagem e desenvolvimento de sistemas orientados a objeto. Estas tentativas iniciais foram recebidas com entusiasmo pelas pessoas envolvidas no desenvolvimento de software orientados a objetos, entretanto, na medida em que grandes diferenças de abordagem se tornaram aparentes, as questões relativas a qual seria o melhor método se tornaram cada vez mais freqüentes ([COL1994]).

Dado que os métodos Booch e OMT estavam crescendo independentemente e sendo reconhecidos pela comunidade usuária como métodos de classe mundial, seus autores, respectivamente, Grady Booch e James Rumbaugh juntaram forças através da Rational Corporation para forjar uma unificação completa de seus trabalhos. Em outubro de 1995, lançaram um rascunho do Método Unificado na versão 0.8, sendo esse o primeiro resultado concreto de seus esforços. Também no outono de 1995, Ivar Jacobson juntou-se à equipe de unificação fundindo o método OOSE. Como autores, Booch, Rumbaugh e Jacobson estavam motivados em criar uma linguagem de modelagem unificada que tratasse assuntos de escala inerentes a sistemas complexos e de missão crítica, que se tornasse poderosa o suficiente para modelar qualquer tipo de aplicação de tempo real, cliente/servidor ou outros tipos de

softwares padrões. Assim surgiu a UML-*Unified Modeling Language*, uma linguagem de modelagem bem definida, expressiva, poderosa e geralmente aplicável ([FUR1998]).

A UML é a linguagem padrão para especificar, visualizar, documentar e construir artefatos de um sistema e pode ser utilizada com todos os processos ao longo do ciclo de vida de um software, tanto na fase de análise e projeto como na fase de desenvolvimento. A UML é uma linguagem de modelagem visual e apresenta alguns benefícios, como melhor visualização dos relacionamentos existentes entre diversos componentes da aplicação, melhor gerenciamento da complexidade, onde cada aspecto do sistema é desenhado à parte em um modelo específico. Porém para fazer uso destes benefícios é necessário uma ferramenta para melhor visualização dos modelos.

Segundo [KIN1995], é importante o uso de uma ferramenta para automatizar o processo de análise e desenvolvimento Orientado a Objetos, para tornar esta fase mais simples, segura e produtiva. Esta ferramenta poderia criar condições para que o analista pudesse realizar o seu trabalho de uma forma mais rápida e dinâmica. Muitos conceitos de Análise Orientada a Objetos poderiam estar embutidos na ferramenta, sem a necessidade do analista lembrá-los a todo momento.

O resultado deste trabalho visa desenvolver uma ferramenta que facilite a análise e desenvolvimento de software. Esta ferramenta permitirá gerar o código fonte em C++ a partir do diagrama de classes da ferramenta Rational Rose C++ 4.0, e ainda gerar o diagrama de classes para a mesma ferramenta, a partir de um código fonte em C++, utilizando-se da engenharia reversa.

A especificação será feita utilizando uma metodologia orientada a objetos, representada através da UML. A ferramenta utilizada para esta especificação será o Rational Rose C++ 4.0, devido aos recursos disponíveis para aplicar as representações da UML, como o Diagrama de Classes e o Diagrama de Casos de Uso.

A ferramenta utilizada para a implementação será o ambiente de desenvolvimento Delphi 5.0, pois já é uma ferramenta bem conhecida e conceituada no mercado.

1.1 OBJETIVOS

O objetivo principal deste trabalho é desenvolver uma ferramenta para criação do diagrama de classes, conforme padrão da UML, onde se possa gerar o código fonte em C++ a partir de um diagrama de classes da ferramenta Rational Rose. Além disso o software deve permitir que a partir de um fonte em C++ se possa gerar o diagrama das classes, aplicando a engenharia reversa em nível de código fonte, para facilitar o processo de análise e desenvolvimento de software.

1.2 ORGANIZAÇÃO

O primeiro capítulo define os objetivos do trabalho, apresentando a justificativa para seu desenvolvimento.

O segundo capítulo apresenta uma visão geral sobre Orientação a Objetos, mostrando a história e conceitos básicos necessários para o entendimento do diagrama de classes da UML. Este capítulo também enfocará a UML, seus modelos, diversos diagramas e história. Para finalizar o capítulo, serão enfocadas as características principais da reengenharia e engenharia reversa, salientando os objetivos, benefícios e tipos de engenharia reversa.

No terceiro capítulo, serão apresentadas as ferramentas envolvidas neste trabalho. Será abordado a ferramenta de modelagem Rational Rose. Será apresentado brevemente as principais características existentes no ambiente de desenvolvimento Delphi 5.0, o qual foi utilizado para desenvolver o protótipo. Será abordado ainda as principais características da linguagem de programação C++.

No quarto capítulo, será apresentado a especificação do protótipo e o seu funcionamento.

O quinto capítulo apresenta as sugestões, dificuldades encontradas e conclusões do trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 ORIENTAÇÃO A OBJETOS

Segundo [FUR1998], o sucesso no desenvolvimento de software depende em grande parte do conhecimento que não só envolve programação e habilidades de gerenciamento, mas também conhecimento e compreensão das mais recentes inovações na indústria de software. De acordo com [COA1991], a programação baseada em objetos foi discutida pela primeira vez no final dos anos sessenta por aqueles que trabalhavam com a linguagem SIMULA. Nos anos setenta, ela era uma parte importante da linguagem Smalltalk desenvolvida pela Xerox PARC. Havia pouca, ou nenhuma, discussão sobre projeto baseado em objetos, e virtualmente nenhuma discussão sobre análise baseada em objetos.

Segundo [RUM1994], a maior parte dos esforços atuais da comunidade que se baseia em objetos tem sido focalizada nos problemas da linguagem de programação. As linguagens de programação baseadas em objetos são úteis para remover as restrições devido à inflexibilidade das linguagens de programação tradicionais.

A vantagem real provém da abordagem dos problemas conceituais iniciais em lugar dos mais tardios problemas de implementação. As falhas de projeto surgidas durante a implementação têm correção mais dispendiosa do que aquelas encontradas mais cedo. O enfoque precoce nos problemas da implementação restringe as opções de projeto e muitas vezes conduz a um produto inferior ([RUM1994]).

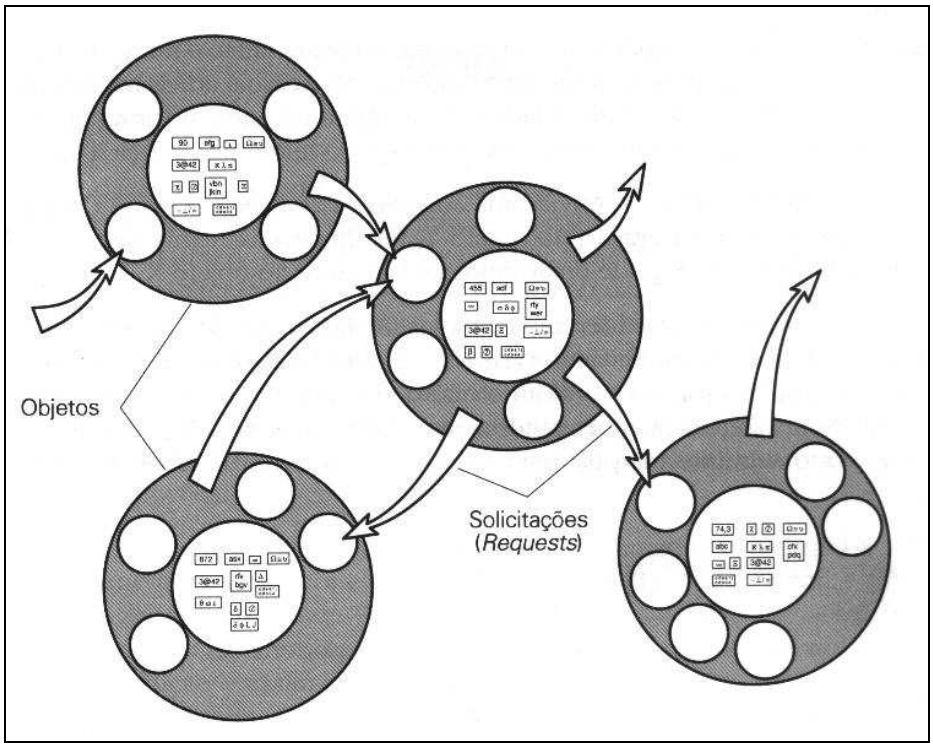
2.1.1 CONCEITOS BÁSICOS

As idéias fundamentais que suportam a tecnologia baseada em objetos incluem:

a) objeto: é qualquer coisa, real ou abstrata, sobre a qual se armazenam dados e operações que manipulam os dados ([RUM1994]). De acordo com [COL1994], um objeto corresponde a uma concepção, abstração ou coisa que pode ser identificada distintamente. Pode-se dizer que objeto é uma instância de uma classe. Um objeto pode ser real ou abstrato, tal como uma fatura, uma organização, uma tela com a qual o usuário interage, um avião, uma reserva de passagem aérea ([MAR1994]. A

figura 1 ilustra a comunicação entre objetos, no circulo mais central estão os atributos, e no circulo mais externo estão os métodos. Para acessar um dado de um objeto, é necessário fazer uma solicitação;

Figura 1 – Comunicação entre objetos



Fonte: [MAR1994]

b) classe: é uma implementação de um tipo de objeto. Ela tem uma estrutura de dados e métodos que especificam as operações que podem ser feitas com aquela estrutura de dados ([MAR1994]). É uma abstração, que representa uma idéia ou noção geral de um conjunto de objetos similares ([COL1994]). Definindo brevemente uma classe é como uma estrutura, excetuando-se o fato de que uma classe pode ser definida contendo funções e dados ([HOL1993]). Pode-se dizer que classe são tipos formais e objetos são as variáveis específicas de cada tipo;

c) métodos: especificam a maneira como as operações são codificadas no software ([MAR1994]). São códigos para implementação em uma classe, ou operação interna, ou seja, o processo de desenvolvimento ([COL1994]);

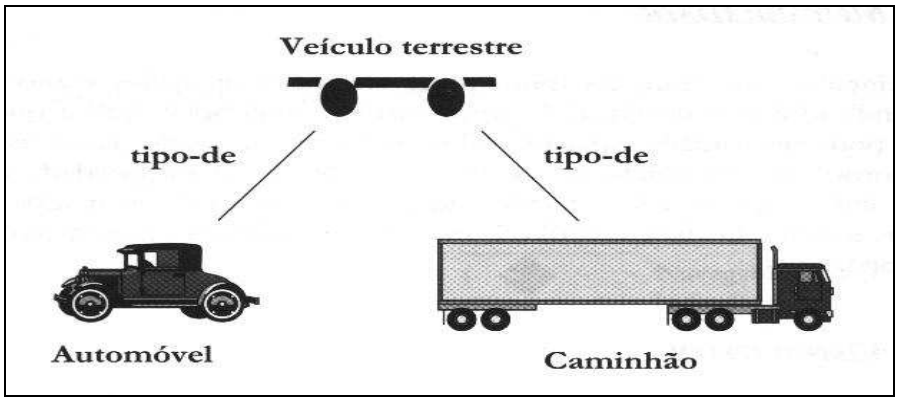
d) solicitações: para fazer com que um objeto faça alguma coisa, é necessário enviar a ele uma solicitação. Essa solicitação faz com que uma operação seja ativada. A operação executa o método adequado e, opcionalmente, devolve uma resposta. ([MAR1994]);

e) encapsulamento: é o resultado (ou o ato) de ocultar, do usuário, os detalhes da implementação de um objeto ([MAR1994]). O encapsulamento esconde dos usuários de um objeto, os detalhes da implementação interna;

f) polimorfismo: é a habilidade de duas ou mais classes responderem à mesma solicitação, cada uma a seu modo ([REE1995]). Métodos que utilizam o polimorfismo usam a mesma expressão para denotar diferentes operações. ([GRA1994]);

g) generalização e herança: a generalização e herança são abstrações poderosas para o compartilhamento de similitudes entre classes, ao mesmo tempo que suas diferenças são preservadas ([MAR1994]). Generalização é o relacionamento entre uma classe e uma ou mais versões refinadas dela. A classe que estiver em processo de refinamento é chamada de superclasse e cada versão refinada é denominada subclasse. A figura 2 ilustra um exemplo de herança através de um exemplo do mundo real, onde automóvel é um tipo de veículo-terrestre e herda todas as suas características e comportamento;

Figura 2 – Exemplo de generalização/herança



Fonte: [FUR1998]

h) associação: uma associação é usada para representar uma dependência estrutural entre objetos, geralmente de classes diferentes ([FUR1998]);

i) agregação: em termos gerais, uma agregação é uma forma especial de associação utilizada para mostrar que um tipo de objeto é composto, pelo menos em parte, de outro em uma relação todo/parte, por exemplo, um pedido é composto por itens de pedido ([FUR1998]);

[MAR1994] resume os muitos benefícios da orientação a objetos em:

a) reaproveitamento: as classes são projetadas de forma que possam ser reutilizadas em muitos sistemas. Para maximizar a reutilização, as classes podem ser construídas de forma que possam ser ajustadas a cada sistema (customizadas). Um dos principais objetivos das técnicas baseadas em objetos é o de se conseguir reaproveitamento em massa na construção de software;

b) estabilidade: as classes projetadas para reutilização repetida tornam-se estáveis da mesma forma que os microprocessadores e outros chips tornam-se estáveis. Os aplicativos devem ser construídos a partir de “chips” de software, sempre que possível;

c) abstração: o projetista pensa em termos de comportamento dos objetos e não em detalhes de baixo nível. O encapsulamento esconde o detalhe e faz com que as classes complexas se tornem fáceis de serem utilizadas. As classes são como caixas-pretas: o profissional de desenvolvimento utiliza a caixa-preta, sem olhar o seu interior. Ele tem que compreender o comportamento da caixa-preta e como se comunicar com ela;

d) confiabilidade: o software construído de classes estáveis, bem testadas e aprovadas, é menos suscetível a erros do que o software inventado do nada. Cada método em uma classe deve ser, por si só, relativamente simples e deve ser projetado para ser confiável;

e) desenvolvimento acelerado: os aplicativos são criados com componentes preexistentes. Muitos componentes são construídos de forma a que possam ser ajustados para um projeto em particular;

f) integridade: as estruturas de dados só podem ser usadas com métodos específicos. Isso é particularmente importante nos sistemas com cliente-servidor e objetos distribuídos**,** onde usuários desconhecidos podem tentar acessar o sistema;

g) programação facilitada: os programas são construídos em partes pequenas, sendo assim, cada uma delas torna-se mais fácil de ser criada em um método. O método transforma o estado dos objetos, de modo normalmente simples, se considerados isoladamente;

h) manutenção facilitada: o programador de manutenção altera um método de uma classe de cada vez. Isto é possível porque cada classe executa suas operações independentemente de outras classes;

i) independência de projeto: as classes devem ser projetadas para serem independentes de plataformas, hardware e ambientes de software. Elas empregam solicitações e respostas de formatos-padrão. Isso permite que as classes sejam utilizadas com diferentes sistemas operacionais, gerenciadores de banco de dados, interfaces gráficas, e assim por diante. O profissional que desenvolve o software não tem que se preocupar com o ambiente ou esperar até que ele seja especificado;

j) interoperabilidade: softwares de vários fornecedores distintos podem operar conjuntamente. Existe uma forma padrão de encontrar classes e de interagir com as classes. A interoperabilidade de software de diferentes fabricantes é um dos principais objetivos dos padrões baseados em objetos. Softwares desenvolvidos independentemente, em lugares separados, deveriam ser capazes de operar conjuntamente e parecer único aos olhos do usuário;

k) processamento cliente-servidor: em sistemas cliente-servidor, as classes, no software cliente, devem enviar solicitações às classes no software servidor e receber respostas. Uma classe do servidor pode ser utilizada por muitos clientes

diferentes. Esses clientes só podem acessar dados do servidor com os métodos da classe. Portanto, os dados destas classes estão protegidos;

l) processamento distribuído em larga escala: as redes internacionais empregarão diretórios de softwares de objetos acessíveis. O projeto baseado em objetos, é a chave para massificação dos sistemas distribuídos. As classes, em uma máquina, interagirão com classes de outro local (objetos distribuídos), sem saber onde elas residem. Elas enviam e recebem mensagens baseadas em objetos, no formato-padrão;

m) processamento paralelo: a velocidade das máquinas será acentuada, e muito, com a construção de computadores paralelos. Processamento concorrente ocorrerá em processadores múltiplos simultaneamente. Os objetos em processadores diferentes serão executados simultaneamente, cada um agindo independentemente. Um *Object Request Broker* (ORB) permitirá que as classes em processadores separados enviem solicitações umas às outras;

n) migração: aplicativos já existentes ou aplicativos não baseados em objetos freqüentemente podem ser preservados, se adaptados com uma embalagem baseada em objetos, de maneira que a comunicação com eles seja feita por mensagens no padrão baseado em objetos;

o) Ferramentas CASE melhores: as ferramentas CASE usam técnicas gráficas para projetar as classes e as interações entre elas e para adaptar objetos já existentes aos novos aplicativos. As ferramentas devem facilitar a modelagem em termos de eventos, gatilhos, estado de objeto, e assim por diante. As ferramentas CASE devem gerar código assim que as classes estejam definidas e devem permitir que o projetista use e teste os métodos criados.

Apesar de todos estes benefícios, a orientação a objetos também possui alguns problemas, principalmente quando se refere ao desenvolvimento de software em equipes de programadores. Segundo [COL1994], a orientação a objetos na forma em que aparece nas linguagens de programação representa apenas uma solução parcial para o problema de

desenvolvimento de software. Apesar de tornar mais fácil o desenvolvimento de programas, o desenvolvimento de um sistema não se restringe apenas à criação de códigos. Seus problemas incluem:

a) ênfase no código: a ênfase comum no desenvolvimento orientado a objetos se localiza nas técnicas e linguagens de programação, e não no processo de desenvolvimento. Quando expressos em termos de programação, os modelos de análise e de projeto não são suficientemente abstratos;

b) trabalho em equipe não levado em conta: sistemas de software são normalmente desenvolvidos por equipes, e não por indivíduos. Orientação a objetos fornece uma ajuda muito pequena para este fato;

c) dificuldade para encontrar os objetos: a definição de objetos e classes corretos em um sistema orientado a objetos não é uma tarefa fácil. Isso se deve, em parte, á falta de familiaridade com o tipo abordagem, como também decorre da dificuldade intrínseca da abordagem;

d) mudanças no gerenciamento: a abordagem orientada a objetos é fundamentalmente distinta da abordagem de decomposição funcional. A introdução de novas sistemáticas de trabalho requer novas formas de gerenciamento;

e) necessidade de intenso treinamento ([MAR1994]) : para utilizar bem a tecnologia baseada em objetos, é preciso treinamento intenso e de boa qualidade. Leva tempo até que os profissionais pensem em termos de encapsulamento, herança e diagramas de análise e projeto baseados em objetos. Após uma tentativa de mudança para técnicas baseadas em objetos, analistas tradicionais talvez ainda tenham tendência a pensar em decomposição estruturada, diagramas de fluxo de dados e em uso tradicional de base de dados.

A solução desse problemas não se restringe apenas ao uso de linguagens mais poderosas, ou a uma melhor formação do programador. São necessários processos de desenvolvimento específicos para a produção de software orientado a objetos ([COL1994]).

Apesar dos problemas citados anteriormente, [MAR1994] afirma que há pouca dúvida de que as técnicas baseadas em objetos vão, eventualmente, permear quase todo o desenvolvimento de software. Apenas os empreendimentos mais ultrapassados não serão baseados em objetos. As empresas que chegarem lá primeiro vão receber os benefícios mais cedo. Porém, sua aceitação vai se espalhar devagar, e muitas organizações vão ter problemas devido à formação inadequada e à falta de talento gerencial.

2.1.2 UMA MUDANÇA NA MANEIRA DE PENSAR

Existem muitos benefícios na tecnologia baseada em objetos e alguns problemas, conforme foi citado anteriormente. Dentre os benefícios, segundo [MAR1994], talvez o mais importante seja a mudança na maneira como as pessoas pensam. Profissionais de sistemas de informação foram todos ensinados a pensar como um computador. Este tipo de pensamento desaba quando o nível de complexidade é alto e, particularmente, quando os computadores usam processamento paralelo em larga escala. A análise baseada em objetos se assemelha ao modo como os humanos categorizam e compreendem seu mundo. As ferramentas CASE-OO permitem gerar código baseado nesse modo mais humano de pensar.

“A medida que os computadores se tornam mais complexos, os humanos não devem ter que pensar como computadores, ao contrário, os computadores devem ser construídos para pensarem como humanos.” ([MAR1994]).

Essa idéia, fundamentalmente diferente, sobre softwares e sistemas vai possibilitar construir sistemas melhores e, particularmente importante, melhorar a comunicação entre usuários finais e analistas. Os profissionais de sistemas de informação e os executivos precisam interagir, formulando juntos modelos de projetos e políticas gerenciais. O raciocínio baseado em objetos permite atingir uma automação mais poderosa de projetos altamente complexos.

A mudança no modo de pensar sobre sistemas é tão fundamental que precisa ser ensinada aos profissionais de sistemas de informação em todos os lugares, nas universidades, escolas técnicas e faculdades de administração. A longo prazo, ela vai mudar toda a profissão

de informática e a maneira como os usuários finais interagem com essa profissão ([MAR1994]).

2.2 UML – THE UNIFIED MODELING LANGUAGE

Este capítulo está focado na semântica de utilização da linguagem de modelagem de objetos unificada: a UML, e como ela aborda os conceitos fundamentais da orientação a objetos.

Segundo [RAT1997], a UML é uma linguagem para especificação, visualização, construção e documentação de artefatos que compõem um sistema, bem como para modelagem de negócios e outras notações. Ela está sendo apoiada por grandes empresas, tais como, IBM, Oracle e outras. A UML pode ser utilizada em todos os processos ao longo do ciclo de desenvolvimento ([FUR1998]).

Segundo [FUR1998], a UML é uma linguagem de modelagem, não uma metodologia. Muitas metodologias consistem, pelo menos em princípio, de uma linguagem de modelagem e um procedimento de uso de linguagem. A UML não prescreve explicitamente esse procedimento de utilização.

O modo para descrever os vários aspectos de modelagem pela UML é através da notação definida pelos seus vários tipos de diagramas. Um diagrama é uma apresentação gráfica de uma coleção de elementos de modelo, freqüentemente mostrado como um gráfico conectado de arcos (relacionamentos) e vértices ([FUR1998]).

Os diagramas utilizados pela UML são compostos de nove tipos: diagrama de caso de uso, de classes, de objeto, de estado, de seqüência, de colaboração, de atividade, de componente e o de execução.

Todos os sistemas possuem uma estrutura estática e um comportamento dinâmico. A UML suporta modelos estáticos (estrutura estática), dinâmicos (comportamento dinâmico) e funcional. A modelagem estática é suportada pelo diagrama de classes e de objetos, que consiste nas classes e seus relacionamentos. Os relacionamentos podem ser de associações, herança (generalização), dependência ou refinamentos. Os modelamentos dinâmicos são

suportados pelos diagramas de estado, seqüência, colaboração e atividade. E o modelamento funcional é suportado pelos diagramas de componente e execução.

Todos os tipo de diagramas da UML serão abordados, porém o diagrama de classes será abordado com maior importância.

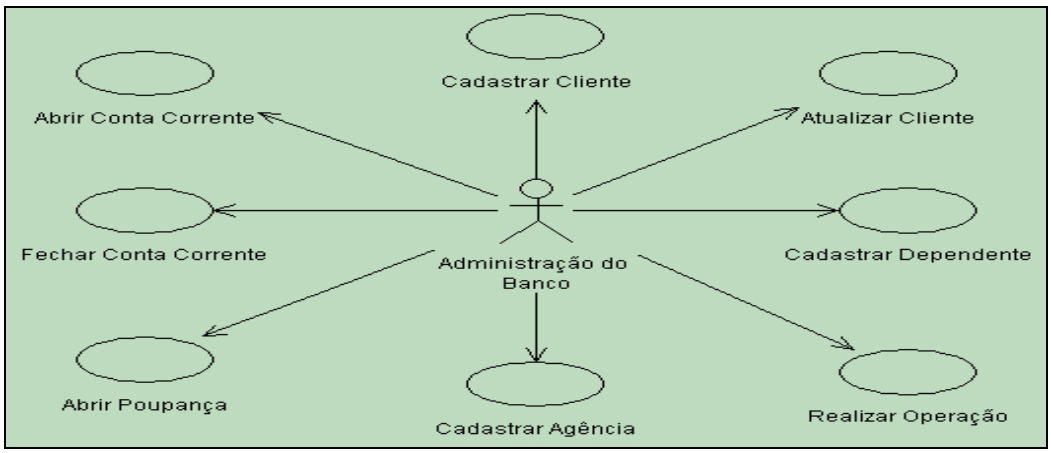
2.2.1 DIAGRAMA DE CASO DE USO

Segundo [FUR1998], os casos de uso descrevem a funcionalidade do sistema percebida por atores externos. Um ator interage com o sistema podendo ser um usuário, dispositivo ou outro sistema.

Segundo [ERI1998], a modelagem de um diagrama de caso de uso é uma técnica usada para descrever e definir os requisitos funcionais de um sistema. Eles são escritos em termos de atores externos, casos de uso e o sistema modelado. Os atores representam o papel de uma entidade externa ao sistema como um usuário, um hardware, ou outro sistema que interage com o sistema modelado.

Atores e casos de uso são classes. Um ator é conectado a um ou mais casos de uso através de associações, e tanto atores quanto casos de uso podem possuir relacionamentos de generalização que definem um comportamento comum de herança em superclasses especializadas em subclasses.

Figura 3 – Diagrama de caso de uso



Fonte: [PAB2000]

O diagrama de casos de uso, mostrado na figura 3, especifica que funções o administrador de uma agência bancária poderá desempenhar. Pode-se perceber que não existe nenhuma preocupação com a implementação de cada uma destas funções, já que este diagrama apenas se resume a determinar que funções deverão ser suportadas pelo sistema modelado.

2.2.2 DIAGRAMA DE CLASSES

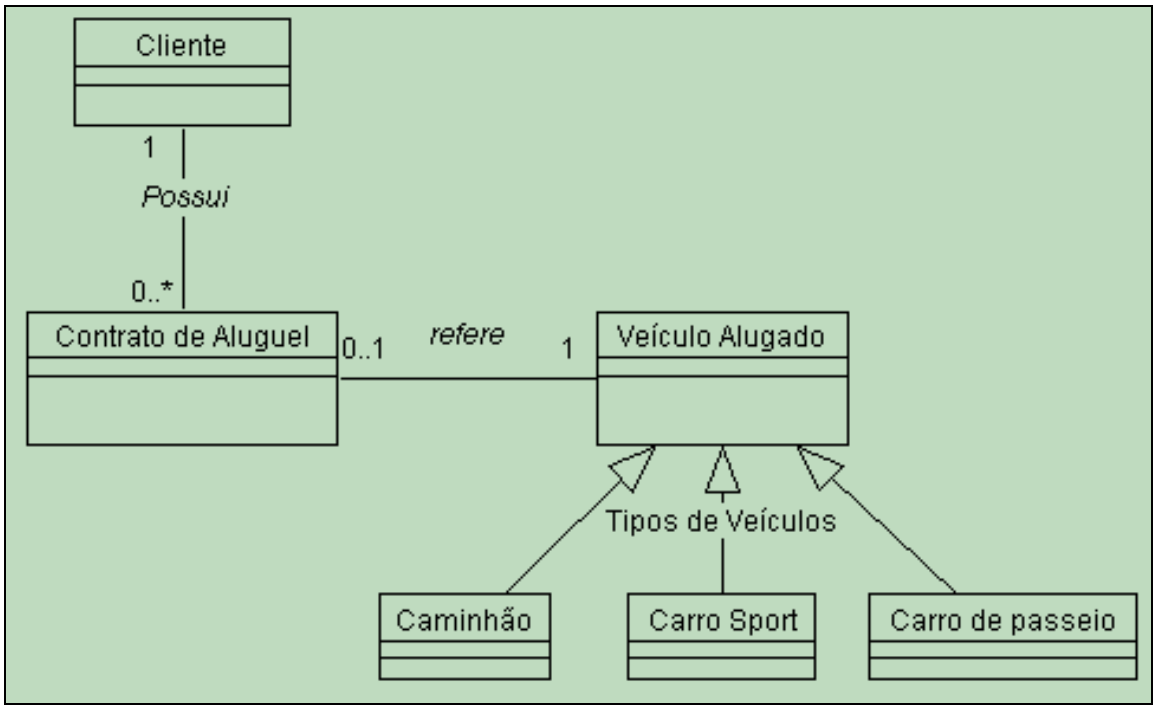
Segundo [FUR1998], o diagrama de classe é a essência da UML, resultado de uma combinação de diagramas propostos pela OMT, Booch e vários outros métodos. Trata-se de uma estrutura lógica estática em uma superfície de duas dimensões mostrando uma coleção de elementos declarativos de modelo, como classes, tipos e seus respectivos conteúdos e relações.

Segundo [ERI1998], o diagrama de classes demonstra a estrutura estática das classes de um sistema onde estas representam as "coisas" que são gerenciadas pela aplicação modelada. Classes podem se relacionar com outras através de diversas maneiras: associação (conectadas entre si), dependência (uma classe depende ou usa outra classe), especialização (uma classe é uma especialização de outra classe), ou em pacotes (classes agrupadas por características similares). Todos estes relacionamentos são mostrados no diagrama de classes juntamente com as suas estruturas internas, que são os atributos e operações. O diagrama de classes é considerado estático já que a estrutura descrita é sempre válida em qualquer ponto do ciclo de vida do sistema.

Conforme [FUR1998] o diagrama de classes é um gráfico bidimensional de elementos de modelagem que pode conter tipos, pacotes, relacionamentos, instâncias, objetos e vínculos (conexão entre dois objetos).

A figura 4 ilustra o diagrama de classes de um sistema de locação de veículos, com relacionamentos entre cliente, contrato de aluguel e veículo.

Figura 4 – Diagrama de Classes



Fonte: [PAB2000]

Segundo [FUR1998], existem quatro tipos principais de relacionamentos no diagrama de classes:

a) Generalização/Especificação: indica relacionamento entre um elemento mais geral e um elemento mais específico (respectivamente, superclasse e subclasse), também conhecido como herança ou classificação;

b) Agregação: usada para denotar relacionamentos todo/parte (por exemplo, um item de nota fiscal é parte de uma nota fiscal);

c) Associação: utilizada para denotar relacionamentos entre classes não correlatas (por exemplo, um cliente pode comprar vários produtos). Na UML, uma associação é definida como um relacionamento que descreve um conjunto de vínculos, onde vínculo é definido como uma conexão semântica entre tuplas de objetos;

d) Dependência: é um relacionamento entre elementos, um independente e outro dependente, onde uma mudança no elemento independente afetará o elemento dependente.

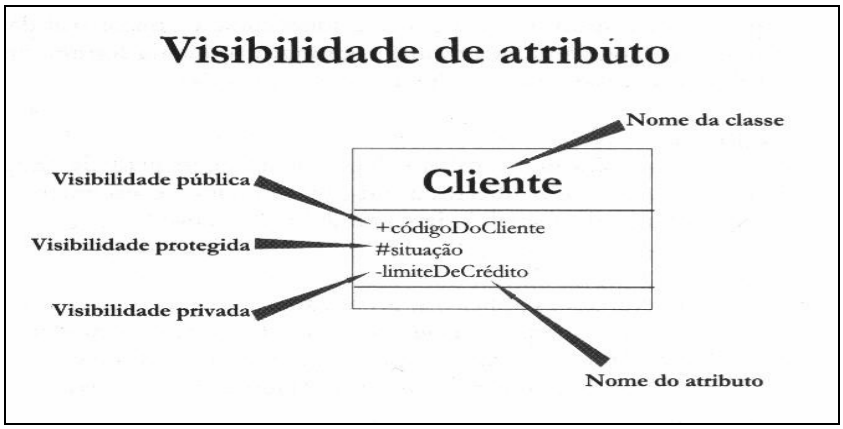
Além dos conceitos de objeto e classe, existem alguns conceitos que são necessários para o entendimento do diagrama de classes da UML:

a) Atributo: atributo é a menor unidade que em si possui significância própria e interrelacionada com o conceito lógico da classe à qual pertence. Na UML os atributos podem ter a seguinte visibilidade:

- + visibilidade pública: significa que todos tem acesso podendo o atributo ser utilizado por operações dentro de outras classe;
- # visibilidade protegida: significa que o atributo é acessado através de operações dentro da mesma classe e por operações de classes ao longo do pacote no qual a classe é definida;
- - visibilidade privada: significa que é parte da interface de uma classe mas não é visível a quaisquer outras classes, o atributo somente pode ser acessado por operações declaradas dentro da mesma classe.

A figura 5 ilustra as três opções de visibilidade dos atributos de uma classe.

Figura 5 – Visibilidade dos atributos de uma classe.



Fonte: [FUR1998]

b) Operação: na UML, um serviço de classe ou comportamento resultante de um procedimento algorítmico é denominado de operação. Há uma distinção importante entre operação e método: uma operação é algo invocado por um objeto enquanto que um método é um corpo de procedimento. Assim um método é uma implementação de uma operação.

Segundo [FUR1998], os nomes das operações são muito importantes: uma operação deve possuir um nome que indique o seu resultado e não os passos executados na obtenção do resultado.

A visibilidade das operações é definida da mesma forma que para os atributos, ou seja, pública, protegida e privada.

c) Associação: associação é uma relação que descreve um conjunto de vínculos entre elementos de modelo. Quando duas ou mais classes, ou mesmo uma classe, apresenta interdependência onde determinada instância de uma classe origina ou se associa a uma ou mais instâncias da outra, dizemos que elas apresentam uma associação;

d) Agregação: a agregação é um caso particular de associação. Indica que uma das classes do relacionamento é uma parte, ou está contida em outra classe. As palavras chaves usadas para identificar uma agregação são: "consiste em", "contém", "é parte de ";

e) Composição: é uma agregação onde uma classe que está contida na outra e constitui a outra. Se o objeto da classe que a contém for destruído, as classes da agregação de composição serão destruídas juntamente.

2.2.2.1 NORMALIZAÇÃO DO MODELO DE CLASSES

Segundo [FUR1998], normalização é um processo formal que examina os atributos de classes com o intuito de minimizar redundância em objetos. Ainda que não esteja explícita na UML, a normalização causa a simplificação de atributos dentro das respectivas classes colaborando para a integridade e a estabilidade do modelo.

A normalização é dependente do paradigma de banco de dados sendo mais necessária para os modelos relacionais, mas modelos de objetos complexos mantém um fundamento algébrico similar em certos aspectos aos modelos relacionais.

A normalização do modelo da classes da UML pode ser dividida em três partes:

a) Primeira Forma Normal: a primeira forma normal é verificada quando em determinadas estruturas de atributos, existem dados que se repetem várias vezes, retratando ocorrências de um mesmo atributo. A primeira forma normal consiste na remoção dos grupos repetitivos de atributos. Podemos tomar como exemplo a criação e normalização do objeto nota fiscal, originado pelo respectivo formulário de nota fiscal, onde os atributos codigoDoProduto e qtdeDoProduto aparecem com ocorrência 15 ocorrências. Aplicando a primeira forma normal sobre a estrutura de atributos considerada, causará a criação da classe agregada [Nota Fiscal, Item] que herdará os atributos repetitivos e da classe [Nota Fiscal].

b) Terceira Forma Normal : pode ocorrer o fato de alguns atributos não serem dependentes diretos do identificador mas, por transitividade, através de outros atributos residentes em uma mesma estrutura referenciada. Segundo [FUR1998], dependência transitiva é a dependência indireta que um determinado atributo tem com o identificador do objeto através de outro atributo explícito ou implícito do qual é diretamente dependente. Como exemplo pode-se citar a classe Nota Fiscal, se nesta classe estiver definido um atributo codigoDoCliente e nomeDoCliente, deverá ser criada uma nova classe para o Cliente e associá-la a nota fiscal.

c) Quarta Forma Normal : tendo como pré-requisito a terceira forma normal, a quarta forma normal busca remover dependências multivaloradas. Para que seja verificada a quarta forma normal deve haver pelo menos três classes envolvidas gerando uma classe de associação.

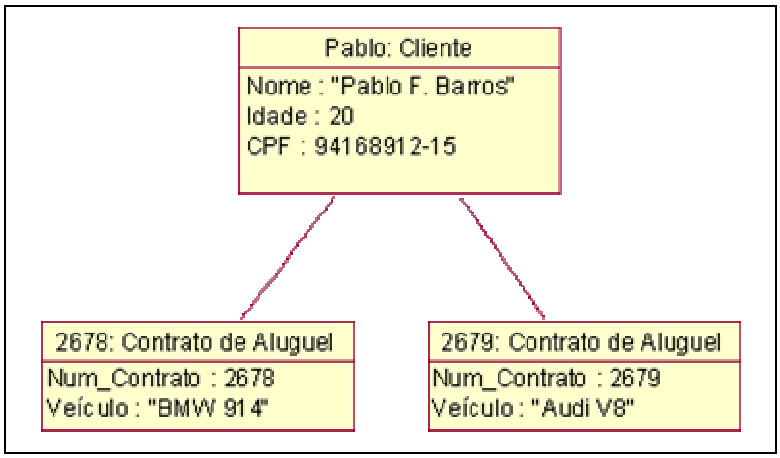
2.2.3 DIAGRAMA DE OBJETOS

Segundo [ERI1998], o diagrama de objetos é uma variação do diagrama de classes e utiliza quase a mesma notação. A diferença é que o diagrama de objetos mostra os objetos que

foram instanciados das classes. O diagrama de objetos é como se fosse o perfil do sistema em um certo momento de sua execução. Os diagramas de objetos não são tão importantes como os diagramas de classes, mas eles são muito úteis para exemplificar diagramas complexos de classes ajudando muito em sua compreensão.

A figura 6 ilustra o diagrama de objetos, onde o cliente possui dois contratos de aluguel de veículos.

Figura 6 – Diagrama de objetos



Fonte: [PAB2000]

2.2.4 DIAGRAMA DE ESTADO

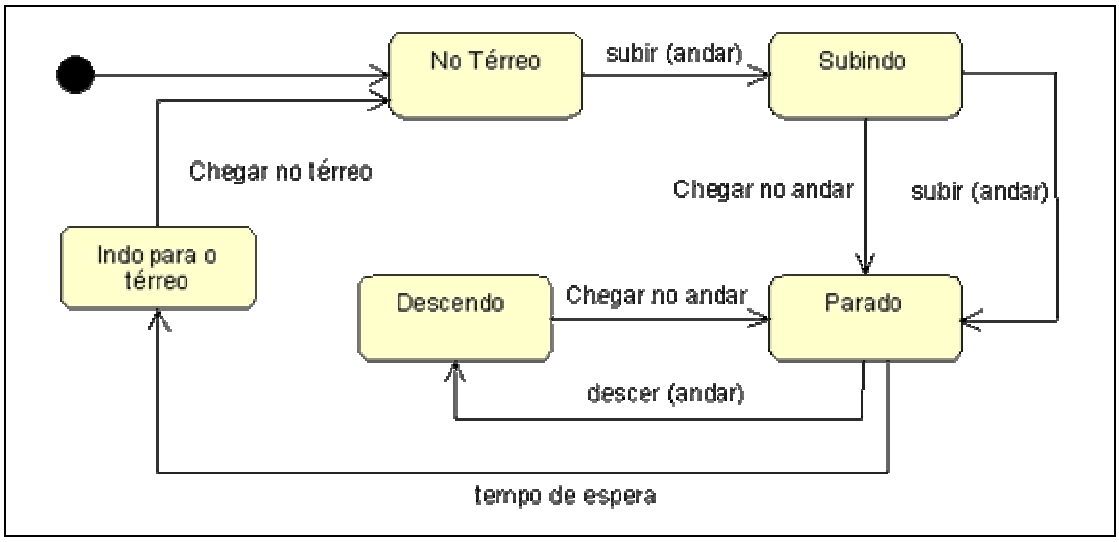
Segundo [ERI1998], o diagrama de estado é tipicamente um complemento para a descrição das classes. Este diagrama mostra todos os estados possíveis que objetos de uma certa classe podem se encontrar e mostra também quais são os eventos do sistemas que provocam tais mudanças. Os diagramas de estado não são escritos para todas as classes de um sistema, mas apenas para aquelas que possuem um número definido de estados conhecidos e onde o comportamento das classes é afetado e modificado pelos diferentes estados.

Conforme [FUR1998], a existência de estado em um objeto implica que a ordem na qual as operações são executadas é importante, o que leva à idéia de objetos como máquinas independentes. Assim, para cada objeto, a ordem das operações no tempo é tão importante que

pode-se formalizar a caracterização do comportamento de um objeto em termos de uma máquina de estado finita equivalente..

A figura 7 ilustra o diagrama de estado com os diversos estados de uma classe elevador.

Figura 7 – Diagrama de estado para uma classe elevador



Fonte: [PAB2000]

Para [ERI1998] os diagramas de estado possuem um ponto de início e vários pontos de finalização. Um ponto de início (estado inicial) é mostrado como um círculo todo preenchido, e um ponto de finalização (estado final) é mostrado como um círculo em volta de um outro círculo menor preenchido. Um estado é mostrado como um retângulo com cantos arredondados. Entre os estados estão as transições, mostrados como uma linha com uma seta no final de um dos estados. A transição pode ser nomeada com o seu evento causador. Quando o evento acontece, a transição de um estado para outro é executada ou disparada.

Uma transição de estado normalmente possui um evento ligado a ela. Se um evento é anexado a uma transição, esta será executada quando o evento ocorrer. Se uma transição não possuir um evento ligado a ela, a mesma ocorrerá quando a ação interna do código do estado for executada (se existir ações internas como entrar, sair, fazer ou outras ações definidas pelo desenvolvedor). Então quando todas as ações forem executadas pelo estado, a transição será disparada e serão iniciadas as atividades do próximo estado no diagrama de estados.

2.2.5 DIAGRAMA DE SEQÜÊNCIA

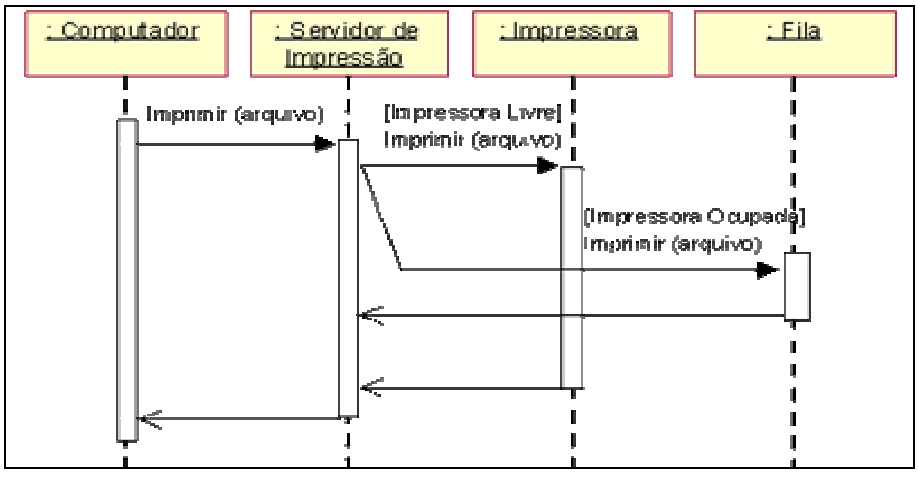
Um diagrama de seqüência mostra a colaboração dinâmica entre os vários objetos de um sistema. O mais importante aspecto deste diagrama é que a partir dele percebe-se a seqüência de mensagens enviadas entre os objetos. Ele mostra a interação entre os objetos, alguma coisa que acontecerá em um ponto específico da execução do sistema. O diagrama de seqüência consiste em um número de objetos mostrado em linhas verticais. O decorrer do tempo é visualizado observando-se o diagrama no sentido vertical de cima para baixo. As mensagens enviadas por cada objeto são simbolizadas por setas entre os objetos que se relacionam ([PAB2000]).

Segundo [FUR1998], diagramas de seqüência possuem dois eixos: o eixo vertical, que mostra o tempo e o eixo horizontal, que mostra os objetos envolvidos na seqüência de uma certa atividade. Eles também mostram as interações para um cenário específico de uma certa atividade do sistema.

No eixo horizontal estão os objetos envolvidos na seqüência. Cada um é representado por um retângulo de objeto (similar ao diagrama de objetos) e uma linha vertical pontilhada chamada de linha de vida do objeto, indicando a execução do objeto durante a seqüência, como exemplo: mensagens recebidas ou enviadas e ativação de objetos.

A figura 8 ilustra o diagrama de seqüência de um processo de impressão de arquivos.

Figura 8 – Diagrama de seqüência



Fonte: [PAB2000]

Os diagramas de seqüência podem mostrar objetos que são criados ou destruídos como parte do cenário documentado pelo diagrama. Um objeto pode criar outros objetos através de mensagens. A mensagem que cria ou destrói um objeto é geralmente síncrona, representada por uma seta sólida

2.2.6 DIAGRAMA DE COLABORAÇÃO

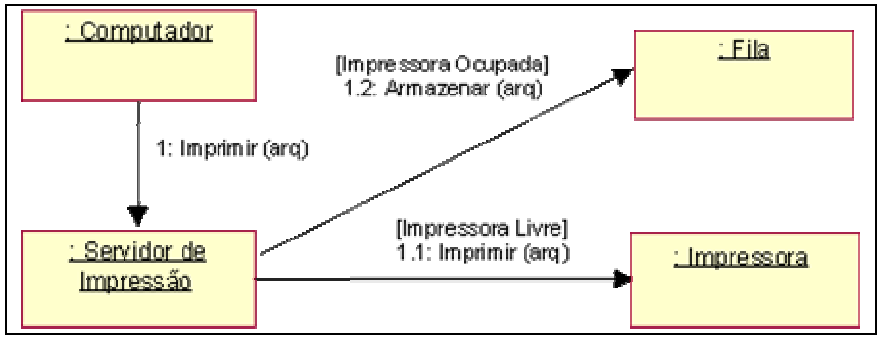
Segundo [FUR1998], um diagrama de colaboração mostra uma interação dinâmica de um caso de uso organizada em torno de objetos e seus vínculos mútuos, de maneira que são usados números de seqüência para evidenciar a seqüência de mensagens.

No diagrama de colaboração, além de mostrar a troca de mensagens entre os objetos, percebe-se também os objetos com os seus relacionamentos. A interação de mensagens é mostrada em ambos os diagramas. Se a ênfase do diagrama for o decorrer do tempo, é melhor escolher o diagrama de seqüência, mas se a ênfase for o contexto do sistema, é melhor dar prioridade ao diagrama de colaboração.

O diagrama de colaboração é desenhado como um diagrama de objeto, onde os diversos objetos são mostrados juntamente com seus relacionamentos. As setas de mensagens são desenhadas entre os objetos para mostrar o fluxo de mensagens entre eles. As mensagens são nomeadas, que entre outras coisas mostram a ordem em que as mensagens são enviadas. Também podem mostrar condições, interações, valores de resposta, e etc.

A figura 9 ilustra o diagrama de colaboração de um processo de impressão de arquivos.

Figura 9 – Diagrama de colaboração



Fonte: [PAB2000]

2.2.7 DIAGRAMA DE ATIVIDADE

Segundo [ERI1998], diagramas de atividade capturam ações e seus resultados. Eles focam o trabalho executado na implementação de uma operação (método), e suas atividades numa instância de um objeto. O diagrama de atividade é uma variação do diagrama de estado e possui um propósito um pouco diferente do diagrama de estado, que é o de capturar ações (trabalho e atividades que serão executados) e seus resultados em termos das mudanças de estados dos objetos.

Os estados no diagrama de atividade mudam para um próximo estágio quando uma ação é executada (sem ser necessário especificar nenhum evento como no diagrama de estado). Outra diferença entre o diagrama de atividade e o de estado é que podem ser colocadas "*swimlanes*". Uma *swimlane* agrupa atividades, com respeito a quem é responsável e onde estas atividades residem na organização, e é representada por retângulos que englobam todos os objetos que estão ligados a ela (*swimlane*) ([ERI1998]).

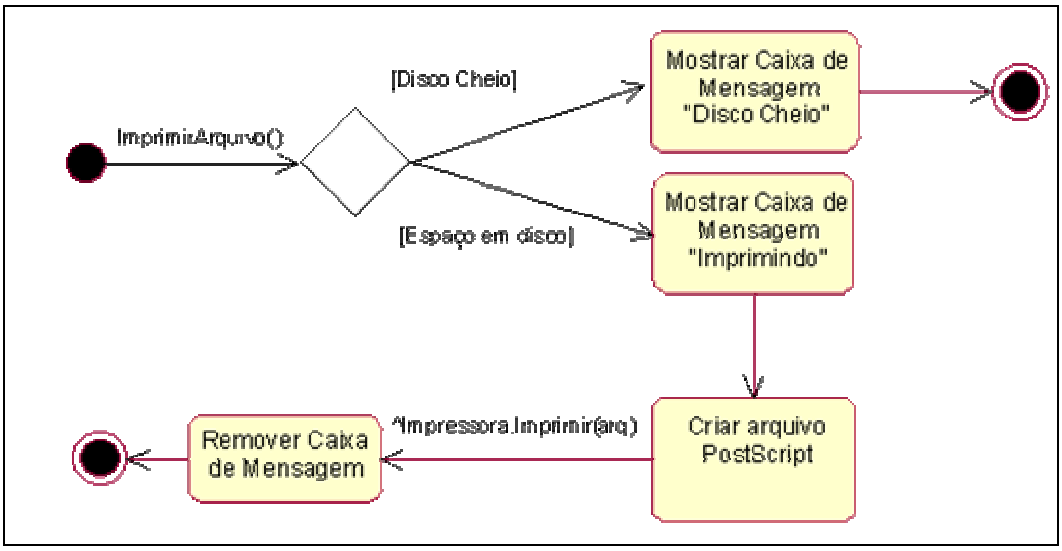
Um diagrama de atividade é uma maneira alternativa de se mostrar interações, com a possibilidade de expressar como as ações são executadas, o que elas fazem (mudanças dos estados dos objetos), quando elas são executadas (seqüência das ações), e onde elas acontecem (*swimlanes*).

Um diagrama de atividade pode ser usado com diferentes propósitos inclusive [ERI1998]:

a) Para capturar os trabalhos que serão executados quando uma operação é disparada (ações). Este é o uso mais comum para o diagrama de atividade;
b) Para capturar o trabalho interno em um objeto;
c) Para mostrar como um grupo de ações relacionadas podem ser executadas, e como elas vão afetar os objetos em torno delas;
d) Para mostrar como uma instância pode ser executada em termos de ações e objetos;
e) Para mostrar como um negócio funciona em termos de trabalhadores (atores), fluxos de trabalho, organização, e objetos (fatores físicos e intelectuais usados no negócio).

A figura 10 ilustra o diagrama de atividade de um processo de impressão de arquivos, mostrando as consistências que são efetuadas durante o processo.

Figura 10 – Diagrama de atividade



Fonte: [PAB2000]

2.2.8 DIAGRAMA DE COMPONENTE

O diagrama de componente é um diagrama que mostra o sistema por um lado funcional, expondo as relações entre seus componentes e a organização de seus módulos durante sua execução.

Segundo [ERI1998], o diagrama de componente descreve os componentes de software e suas dependências entre si, representando a estrutura do código gerado. Os componentes são a implementação na arquitetura física dos conceitos e da funcionalidade definidos na arquitetura lógica (classes, objetos e seus relacionamentos). Eles são tipicamente os arquivos implementados no ambiente de desenvolvimento. Um componente é mostrado na UML como um retângulo com uma elipse e dois retângulos menores do seu lado esquerdo. O nome do componente é escrito abaixo ou dentro de seu símbolo.

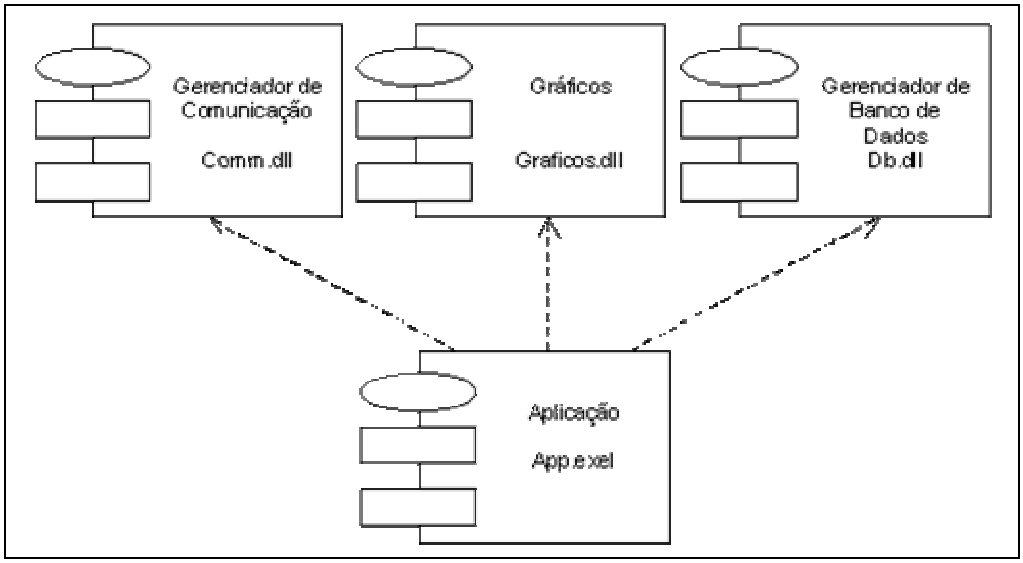
Componentes são tipos, mas apenas componentes executáveis podem ter instâncias. Um diagrama de componente mostra apenas componentes como tipos. Para mostrar instâncias

de componentes, deve ser usado um diagrama de execução, onde as instâncias executáveis são alocadas em *nodes*.

Segundo [ERI1998], a dependência entre componentes pode ser mostrada como uma linha tracejada com uma seta, simbolizando que um componente precisa do outro para possuir uma definição completa. Com o diagrama de componentes é facilmente visível detectar que arquivos *.dll* são necessários para executar a aplicação.

A figura 11 ilustra o diagrama de componente de uma aplicação que interage com diversos componentes externos (dll), cada qual com a sua função específica.

Figura 11 – Diagrama de componente



Fonte: [PAB2000]

Componentes podem definir interfaces que são visíveis para outros componentes. As interfaces podem ser tanto definidas ao nível de codificação (como em Java) quanto em interfaces binárias usadas em *run-time* (como em OLE). Uma interface é mostrada como uma linha partindo do componente e com um círculo na outra extremidade. O nome é colocado junto do círculo no final da linha. Dependências entre componentes podem então apontar para a interface do componente que está sendo usada [ERI1998].

2.2.9 DIAGRAMA DE EXECUÇÃO

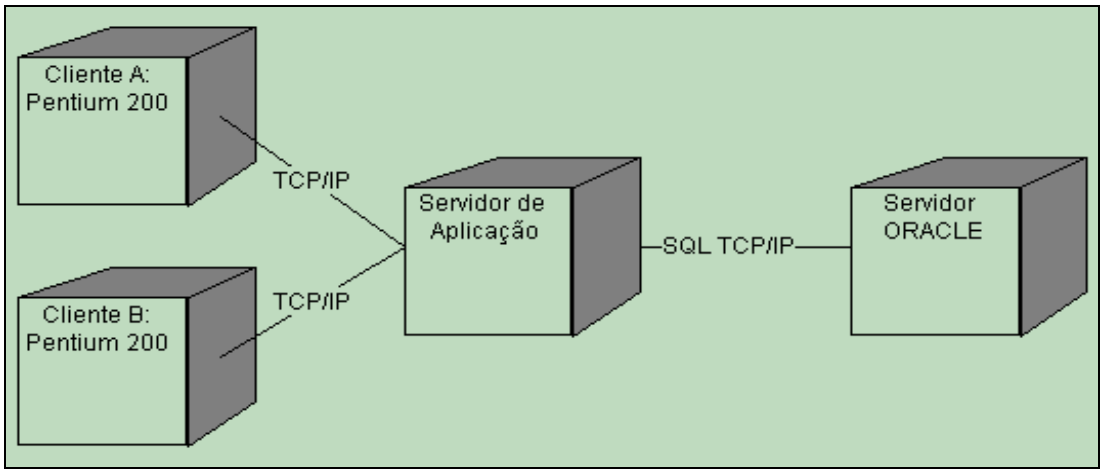
Segundo [ERI1998], o diagrama de execução mostra a arquitetura física do hardware e do software no sistema. Pode mostrar os atuais computadores e periféricos, juntamente com as conexões que eles estabelecem entre si e pode mostrar também os tipos de conexões entre esses computadores e periféricos. Especifica-se também os componentes executáveis e objetos que são alocados para mostrar quais unidades de software são executados e em que destes computadores são executados.

O diagrama de execução demonstra a arquitetura *run-time* de processadores, componentes físicos (*devices*), e de software que rodam no ambiente onde o sistema desenvolvido será utilizado. É a última descrição física da topologia do sistema, descrevendo a estrutura de hardware e software que executam em cada unidade.

O diagrama de execução é composto por componentes, que possuem a mesma simbologia dos componentes do diagrama de componentes, *nodes*, que significam objetos físicos que fazem parte do sistema, podendo ser uma máquina cliente numa LAN, uma máquina servidora, uma impressora, um roteador, etc., e conexões entre estes *nodes* e componentes que juntos compõem toda a arquitetura física do sistema ([ERI1998]).

A figura 12 ilustra o diagrama de execução, onde dois clientes (A e B) fazem solicitações para o servidor de banco de dados Oracle através de um servidor de aplicação.

Figura 12 – Diagrama de execução



Fonte: [PAB2000]

2.3 REENGENHARIA E ENGENHARIA REVERSA

Segundo [FUR1994], Reengenharia da Informação é um conjunto de técnicas e ferramentas orientadas à avaliação, reposicionamento e transformação de sistemas de informação existentes, com o objetivo de estender-lhes a vida útil e, ao mesmo tempo, proporcionar-lhes uma melhor qualidade técnica e funcional.

Ainda conforme [FUR1994], para colocar esses objetivos dentro de uma perspectiva, deve-se analisar com mais detalhes a prática de manutenção atual. Basicamente, a manutenção de sistemas pode ser descrita como um processo que garanta a programas e/ou sistemas a sua continuidade de execução ou que melhore suas características funcionais. A operação envolve a correção de erros, revisão da necessidade e ampliação de funções/melhoria de desempenho, num rateio percentual de:

a) 20% para manutenção corretiva (correção de erros);
b) 25% para manutenção adaptativa (revisão de necessidades);
c) 55% para manutenção aprimorativa (ampliação de funções ou melhoria de desempenho).

A questão básica nesse ponto fica sendo, quando dever-se-ia optar por refazer um sistema em vez de aplicar a reengenharia, ou vice-versa. Primeiramente, devemos atentar para as estatísticas de manutenção de sistemas (se houver) que podem ser úteis na determinação do estado do sistema, incluindo características sobre:

a) tamanho em termos de linhas de código e pontos de função;
b) idade do programa/sistema;
c) Linguagem e modalidade de processamento (batch/on-line);
d) ambiente operacional;
e) tempo e custo de manutenção anual;
f) índice de falhas de produção e número de erros pendentes;
g) número de requisições de manutenção atendidas e pendentes a cada ano;
h) grau de satisfação do usuário.

A lista acima pode ser útil na determinação da posição do sistema, porém, mais importante ainda são a integridade dos dados e a complexidade e o grau de estruturação dos programas.

Algumas das razões para se optar por reengenharia ou pela engenharia são [FUR1994]:

a) razões para reengenharia:
   - freqüentes falhas de produção;
   - problemas de desempenho;
   - tecnologia obsoleta;
   - problemas de integração de sistemas;
   - qualidade técnica ruim;
   - dificuldades para testar e caro para manter;
   - problemas crescentes no sistema;
b) razões para se refazer o sistema:
   - não confiável;
   - algoritmos ruins ou incorretos;
   - não atende as necessidade dos usuários.

2.3.1 OBJETIVOS DA REENGENHARIA

Os principais objetivos da reengenharia são [MCC1993]:

a) identificar e mensurar os sistemas existentes;
b) auxiliar na administração dos sistemas existentes;
c) fornecer respostas para as dúvidas de manutenção;
d) preservar investimentos feitos anteriormente;
e) recuperar e desenvolver padrões;
f) reutilizar componentes dos sistemas existentes.

Existem alguns tipos de reengenharia. Entre eles pode-se citar reestruturação, migração e engenharia reversa.

2.3.2 ENGENHARIA REVERSA

Segundo [FUR1994] engenharia reversa é o processo de derivar as especificações lógicas dos componentes do sistema a partir de sua descrição física, com o auxílio de ferramentas automatizadas. A engenharia reversa trata os dados e processos, apesar de que ultimamente as ferramentas tem sido mais efetivas para aqueles do que para estes.

Conforme [MCC1993], engenharia reversa é o processo de examinação de níveis físicos de software, como código fonte ou descrição de dados/arquivos para recuperar ou reconstruir informações do nível de especificação, descrevendo o processo e dados do software com o auxílio de ferramentas automatizadas.

Como benefícios da engenharia reversa pode-se destacar [FUR1994]:

a) melhor compreensão dos sistemas existentes;
b) fornecimento automático de documentação atualizada dos sistemas existentes;
c) fornecimento de um meio eficiente para análise de dados e processos;
d) aceleração do processo de manutenção de sistemas;
e) agilização da conversão de sistemas e diminuição dos esforços de migração;
f) possibilidade de manutenção de sistemas em nível de desenho.

Os benefícios da engenharia reversa, segundo [MCC1993], além dos citados acima são:

a) posicionamento dos sistemas para serem suportados por ferramentas CASE;
b) recuperação do conhecimento sobre os sistemas existentes;
c) nivelamento do investimento feito nos sistemas existentes.

Tipicamente, a entrada para a engenharia reversa são o código-fonte, o dicionário de dados e as DDLs (*Definition Data Language*) ([FUR1994]). O resultado, por sua vez, dependerá diretamente da ferramenta utilizada, porém, alguns desses produtos deveriam ser gerados:

a) pelo lado dos dados:
   - desenho do banco de dados físico;

- estrutura física de dados;
- diagrama de entidades e relacionamentos;
- modelo de dados normalizado.

b) Pelo lado dos processos:
   - especificação do desenho físico;
   - diagrama de estrutura.

Engenharia reversa não significa a utilização de ferramentas automatizadas que fazem tudo por si próprias através de um simples “apertar de botões” ([FUR1994]). Apesar de existirem boas ferramentas, a reengenharia continua sendo uma atividade trabalhosa.

São candidatos à engenharia reversa as aplicações que não foram projetadas utilizando-se técnicas de modelagem funcional ou de dados, ou aplicações nas quais a documentação original não foi atualizada juntamente com o código-fonte. Segundo [FUR1994], a engenharia reversa também pode ser empregada para apresentar modelos lógicos que representam o que a aplicação faz, através do uso de ferramentas que permitem sumarizar milhares de linhas de código numa simples imagem gráfica.

3 AMBIENTE DE DESENVOLVIMENTO

Neste capítulo serão apresentados os conceitos de ferramentas CASE, e, também as ferramentas utilizadas neste trabalho, tais como, Rational Rose, Borland Delphi 5.0 e alguns conceitos da linguagem C++.

3.1 FERRAMENTAS CASE

Uma ferramenta CASE (*Computer Aided Software Engineering* , ou Engenharia de Software Auxiliada por Computador) é, por definição, uma ferramenta de apoio ao processo de desenvolvimento de software, não a solução mágica para os problemas da área.

Segundo [GAN1990], o termo foi criado no começo dos anos oitenta, quando a idéia de que ferramentas gráficas, como os diagramas de fluxo de dados (DFD), diagramas de entidade-relacionamento (MER ou DER) e gráficos de estrutura poderiam ser úteis em análise e projeto de sistemas.

Segundo [JOA1993], CASE é a automação da automação, e fornece uma resposta prática aos problemas de produtividade e é também uma combinação de ferramentas de software com a metodologia. Deste modo, CASE é diferente das primeiras tecnologias de desenvolvimento de software, porque não enfoca somente a fase de implementação de sistemas.

As primeiras ferramentas CASE objetivaram apenas a automação do desenho de diagramas e o armazenamento de informações básicas desses diagramas. Posteriormente, essas ferramentas passaram por uma revolução, assim como também o conceito de desenvolvimento de software. A partir daí, tais ferramentas passaram a abranger os conceitos de ciclo de vida, dicionário de dados e checagem automática da especificação, surgindo assim, as ferramentas que hoje estão no mercado ([FUG1993]).

Conforme [JOA1993], não existe uma classificação rígida para ferramentas CASE, mas pode-se separá-las nos seguintes grupos:

a) *Front End* ou *Upper Case:* são aquelas que apoiam as etapas iniciais de criação do sistema;

b) *Back End* ou *Lower Case:* são aquelas que apoiam a geração de código e os testes, isto é, a parte referente a implementação do sistema;

c) I-CASE ou CASE Integrado: são aquelas que apoiam todo o ciclo de vida do software.

3.1.1 I-CASE E GERADORES DE CÓDIGO

Segundo [MAR1994], o termo I-CASE significa CASE integrado, onde as ferramentas para todos os estágios do desenvolvimento se unem e governam um gerador de código. I-CASE se refere a um pacote de ferramentas CASE que suporta todos os estágios do ciclo de vida de desenvolvimento, inclusive o de geração de código. [MAR1994] afirma que sempre que possível, os programas devem ser gerados automaticamente a partir de projetos de alto nível, de especificações ou de imagens em uma tela CASE.

Com ferramentas baseadas em objetos, assim que um objeto é descrito para a ferramenta, ela consegue criar tabelas que permitem que instâncias dos objetos possam ser criadas. Uma descrição de seus dados pode ser construída e as tabelas de instâncias, preenchidas. Os métodos podem ser criados e o comportamento dos objetos, observado e ajustado ([MAR1994]).

Tão logo o projetista descreva os tipos de objetos para as ferramentas, os objetos estão lá, e podem ser experimentados. Parece-se com uma planilha eletrônica, na qual o projetista pode acrescentar colunas e linhas, especificar cálculos, imediatamente acrescentar valores e rodar a planilha, gerando e modificando gráficos. Essa capacidade das planilhas de criar resultados instantâneos deveria ser emulada pelas ferramentas CASE-OO.

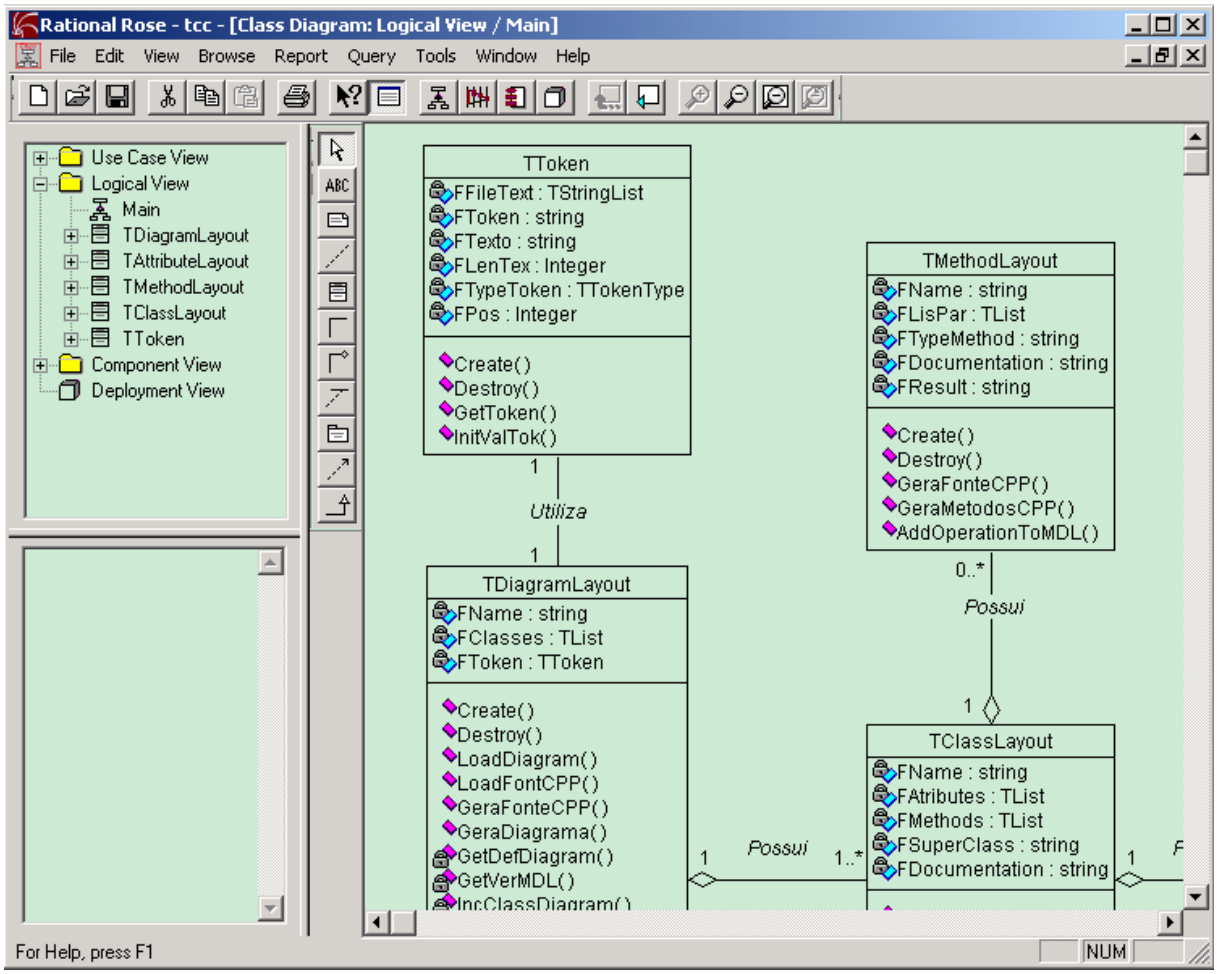
3.1.2 FERRAMENTA RATIONAL ROSE

O *Rational Rose* é uma ferramenta para análise, modelagem, projeto e construção de sistemas orientados a objeto. Dentre os diagramas suportados pelo Rose destacam-se o Diagrama de Casos de Uso, o Diagrama de Classes e o Diagrama de seqüência.

O ROSE é uma das ferramentas de modelagem orientadas a objeto mais interativas do mercado, possuindo uma interface bem amigável. Além de suportar a UML, também é possível modelar utilizando os métodos Booch e OMT.

A figura 13 ilustra o a tela principal da ferramenta Rational Rose, com um diagrama de classes aberto.

Figura 13 – Tela Principal do Rational Rose 4.0



3.2 O AMBIENTE DELPHI 5.0

Escolheu-se este ambiente de desenvolvimento de software pelo fato de que o mesmo atende aos objetivos do trabalho, disponibilizando uma série de componentes e classes que podem ser utilizadas a aprimoradas para o trabalho proposto.

Pode-se citar algumas vantagens do ambiente de programação Delphi:

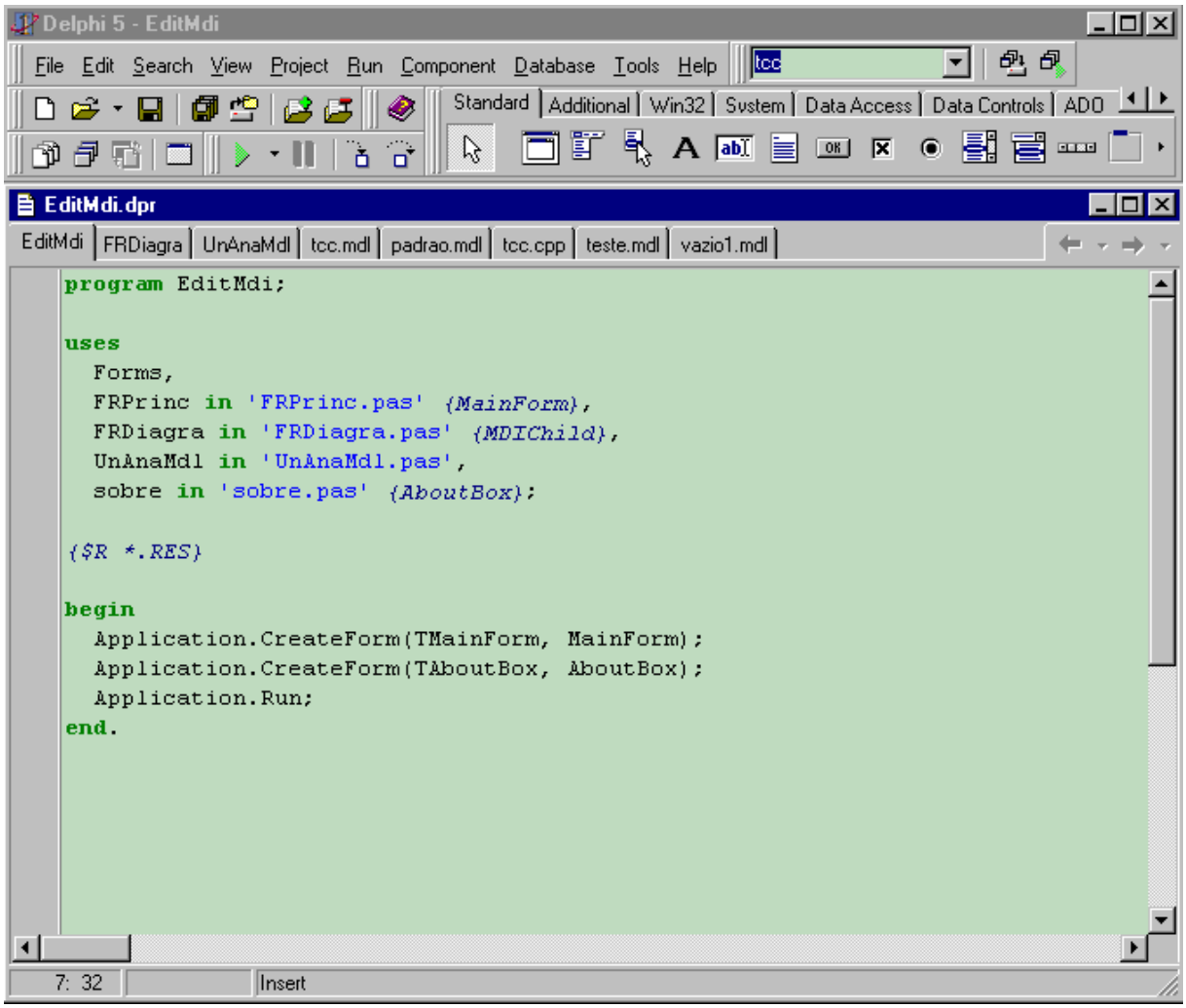
a) gera programas executáveis, que encapsulam todas as bibliotecas necessárias para a sua execução, não sendo necessário ao desenvolvedor carregar as várias bibliotecas;

b) possibilita que seja feita interface do sistema para o usuário;

c) possui boa documentação de auxílio, tendo um tutorial explicativo sobre a ferramenta;

d) facilidade de adaptação para antigos programadores Pascal.

Figura 14 – Tela Principal do Delphi 5



Como pode ser visto na figura 14, o ambiente de Desenvolvimento Delphi é formado por quatro partes principais:

a) a janela principal, na parte superior tem-se o menu, as barras de ferramentas e as paletas contendo os componentes visuais (*Edit, Label*) e não visuais (*Table, Query*, etc);

b) o *Form* é a tela onde o desenvolvedor pode trabalhar no desenvolvimento da interface do aplicativo;

c) o *Code Editor* é a área de desenvolvimento das rotinas e procedimentos (código-fonte). Algum código o Delphi coloca automaticamente quando é inserido algum componente no *Form*.

d) o *Object Inspector* é uma janela dividida em *Properties* (Propriedades) e *Events* (Eventos), onde pode-se visualizar e modificar as propriedades e os eventos de um determinado componente.

3.3 A LINGUACEM C++

Segundo [HOL1993] o C++ é um superconjunto do C. Ele foi originalmente uma extensão do C (seu primeiro nome foi C com classes). O que significa que para aprender C++ deve-se antes conhecer um pouco da linguagem C. O C++ é uma linguagem rica em novos métodos e técnicas poderosas.

A linguagem C++ foi desenvolvida em 1983 por Bjarne Stroustrup. Em essência o C++ é muito semelhante ao C, mas com um número de extensões importantes. A mais importante inovação do C++ em relação ao C é a idéia de objeto.

Apesar do C++ ter sido desenvolvido para introduzir a modularidade em larga escala na linguagem C, existem outros aspectos do C++ que o tornam mais atrativo. Um desse aspectos é a conhecida flexibilidade; pode-se simplesmente redefinir quase todos os operadores do C no C++, bem como fazer coisas extraordinárias com funções. Por exemplo, através da função de sobrecarga, pode-se chamar a mesma função com parâmetros de diferentes tipos. O C++ decidirá qual versão da função usar baseado no tipo de parâmetro passado ([HOL1993]).

O quadro 1 mostra a estrutura básica de um programa fonte na linguagem C++

Quadro 1 - Exemplo de programa fonte C++

```
/* Programa fonte C++ */
#include <iostream.h>

class animal {
public:
      void eat (void);
      void sleep (void);
      void breathe (void);
};

/* Classe elefante herdada de animal */
class elefante : public animal {
public:
      void trumpet (void);
      void stampede (void);
} jumbo;


/* implementação */

void animal::eat (void)
{
    count << “eating...\n”;
}


...
/* programa principal */

main ()

{

      jumbo.breathe ();

      jumbo.trumpet ();

      jumbo.breathe ();

      return (0)

}
```

Fonte:[HOL1993]

4 PROTÓTIPO

Neste capítulo serão apresentados a especificação e metodologia utilizada para o desenvolvimento do protótipo, o seu desenvolvimento e por fim o funcionamento deste.

Como citado anteriormente, o objetivo deste trabalho é desenvolver uma ferramenta para criação do Diagrama de Classes, segundo a UML, onde se possa gerar o código fonte em C++, a partir de um Diagrama de Classes da ferramenta Rational Rose. Além disso o software irá permitir que a partir de um fonte em C++ se possa gerar o diagrama das classes, aplicando a engenharia reversa em nível de código fonte.

4.1 ESPECIFICAÇÃO DO PROTÓTIPO

Para a especificação do protótipo foi adotada a análise orientada a objetos, pois o protótipo foi desenvolvido através da programação orientada a objetos, através do Delphi 5 utilizando a linguagem Object Pascal. Para a especificação das classes foi utilizada a linguagem de modelagem UML, através da ferramenta Rational Rose C++ 4.0.

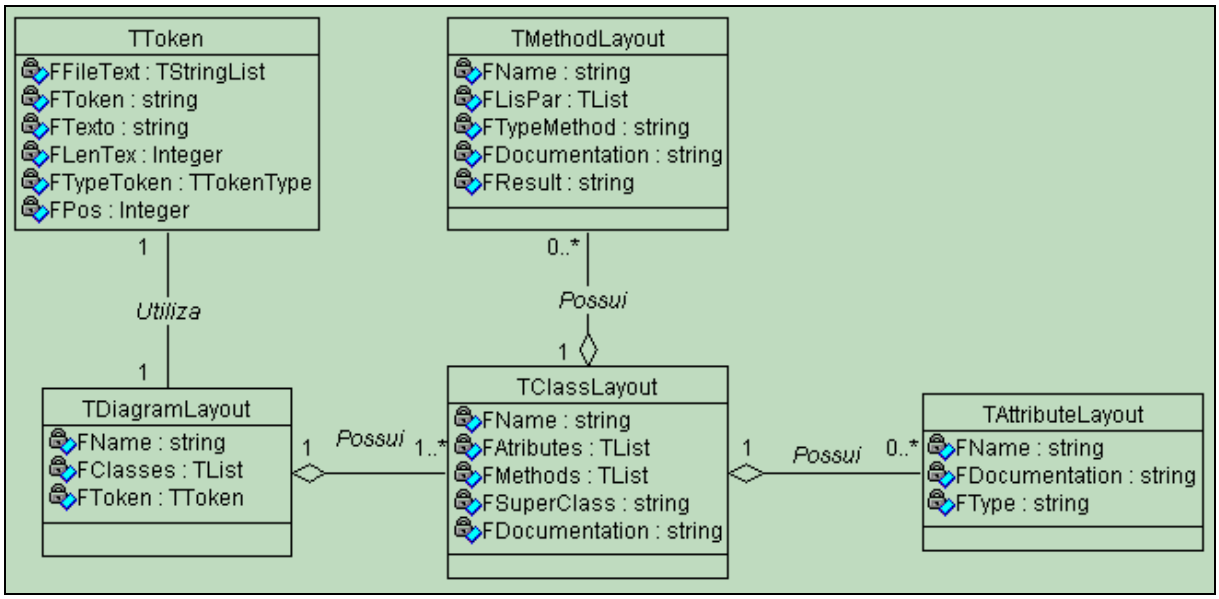
Este protótipo oferece uma maneira simples de gerar código C++ e também de extrair diagrama de classes a partir de um fonte C++, obtendo documentação de um sistema já existente.

O protótipo é basicamente construído pelas seguintes classes: TToken, TDiagramLayout, TClassLayout, TMethodLayout e TAttributeLayout. Todas estas classes serão melhor detalhadas a seguir.

Em nível de telas, o protótipo foi construído utilizando o conceito de MDI-*Multiple Document Interface*, onde cada nova tela aberta é uma tela filha da tela principal. Cada tela que é aberta contém um objeto do tipo TDiagramLayout, para o qual é carregado toda a estrutura do diagrama ou fonte C++ no momento da abertura do arquivo.

A figura 15 mostra a estrutura de classes utilizada para a construção do protótipo, mostrando apenas os atributos de cada classe.

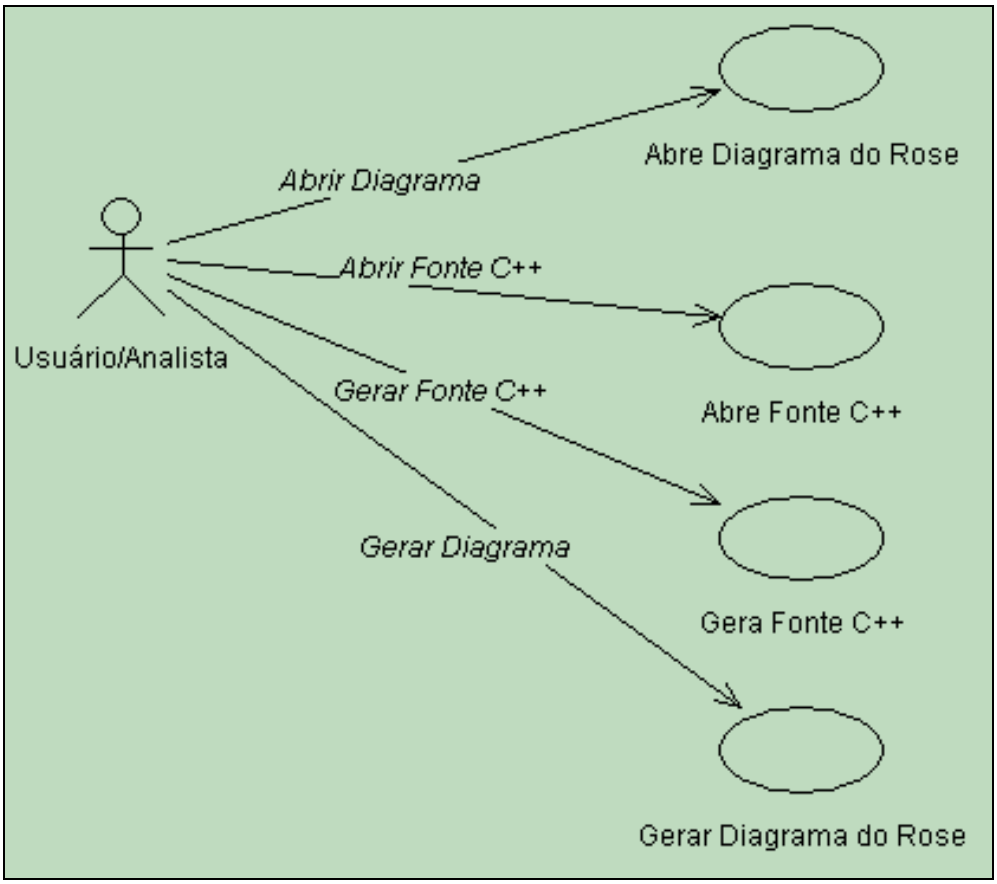
Figura 15 – Esquema de Classes do Protótipo



A classe TDiagramLayout contém uma lista de todas as classes lidas do diagrama ou do arquivo C++. Dentro desta classe existe um atributo FClasses, que é uma lista de Objetos do tipo TClassLayout. A classe TClassLayout, contém dois atributos muito importantes: FAtributes, que é uma lista de Objetos do tipo TAtributeLayout e FMethods, que uma lista de objetos do tipo TMethodLayout.

A seguir está representado o diagrama de caso de uso, outro diagrama importante da UML. A figura 16 ilustra o diagrama de casos de uso do protótipo.

Figura 16 – Diagrama de caso de uso do protótipo.



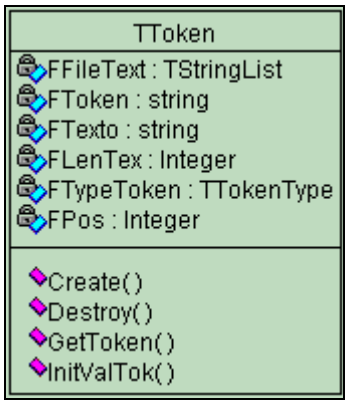
A seguir será explicado o funcionamento de cada classe, com os seus métodos e atributos.

4.1.1 CLASSE TTOKEN

Esta classe é a responsável pela leitura das palavras (Tokens) um a um do arquivo de Diagrama do Rose ou do arquivo fonte em C++. A classe TDiagramLayout utiliza a função GetToken para obter as palavras e analisar o arquivo.

A figura 17 ilustra os principais atributos e métodos da classe TToken.

Figura 17 – Classe TToken.



As propriedades e métodos da classe TToken estão detalhadas no quadro 2.

Quadro 2 – Propriedades/métodos da classe TToken

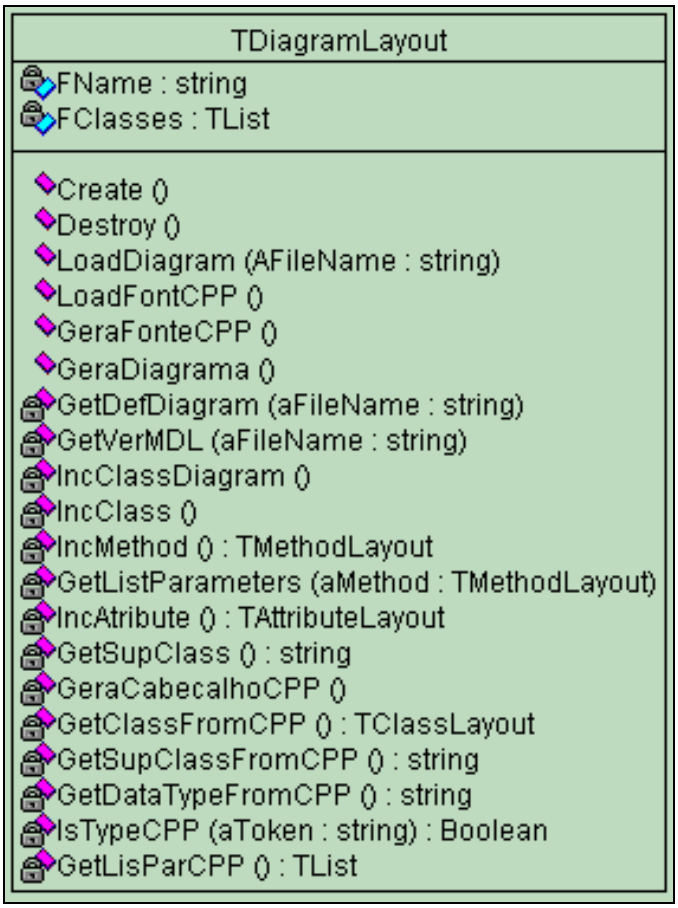
Propriedade/Método	Valor/Ação
FFileText	É uma TStringList com o texto do arquivo selecionado
FToken	Contém a palavra (string) retornada pela função GetToken
FTexto	Ë uma string que contém todo o texto do arquivo.
FLexTex	Contém o tamanho da string FTexto
FTypeToken	Possui o tipo do Token retornado pela função GetToken. Pode ser: (ttIdentify, ttReserved, ttNumber, ttSymbol, ttError, ttIgnore, ttEnd)
FPos	Contém a última posição lida por GetToken na string Ftexto
Create	Função construtora da classe
Destroy	Função destrutora da classe
GetToken	Função responsável pela leitura das palavras (Tokens) de FTexto
InitValTok	Função responsável por inicializar as variáveis internas da classe

4.1.2 CLASSE TDIAGRAMLAYOUT

Esta classe é a principal classe do protótipo. Ela é responsável pela leitura do diagrama do Rose, leitura do arquivo fonte C++, geração do arquivo fonte C++ e geração do diagrama para o Rose. Após o arquivo lido (diagrama ou fonte C++) esta classe conterá toda a estrutura do arquivo, contendo uma lista das classes.

A figura 18 ilustra os atributos e métodos da classe TDiagramLayout.

Figura 18 – Classe TDiagramLayout.



As propriedades e métodos da classe TDiagramLayout são detalhados no quadro 3.

Quadro 3 – Propriedades/métodos da classe TDiagramLayout

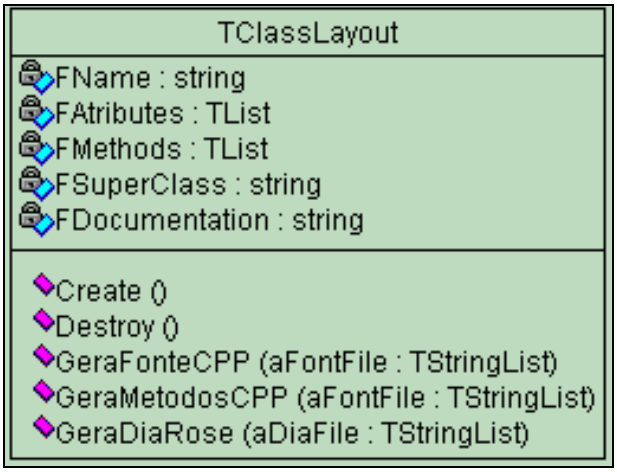
Propriedade/Método	Valor/Ação
FName	Nome do diagrama de classes ou arquivo C++
FClasses	Contém uma lista de objetos do tipo TClassLayout
Create	Função construtora da classe
Destroy	Função destrutora da classe
LoadDiagram	Função responsável pela leitura do diagrama de classes do Rose
LoadFontCPP	Função responsável pela leitura do arquivo fonte C++
GeraFonteCPP	Função responsável por gerar o arquivo fonte C++
GeraDiagrama	Função responsável por gerar o diagrama de classes para o Rose
GetDefDiagram	Interpreta o diagrama de classes e cria lista de classes
GetVerMDL	Verifica a versão do arquivo do Rose. Retorna uma exceção se a versão for diferente de 4.0
IncClassDiagram	Lê informações a respeito do diagrama de classes (Nome)
IncClass	Lê todas as informações de uma classe (Nome, Herança)

IncMethod	Lê a definição de um método do diagrama e retorna uma variável do tipo TMethodLayout
GetListParameters	Verifica a lista de parâmetros do método
IncAtribute	Busca a definição do atributo da classe.
GetSupClass	Busca a classe pai da classe, se houver
GeraCabecalhoCPP	Gera o cabeçalho do arquivo fonte C++
GetClassFromCPP	Lê as informações de uma classe de um arquivo fonte C++
GetSupClassFromCPP	Verifica a classe pai em um arquivo fonte C++
GetDataTypeFromCPP	Verifica o tipo do atributo de uma classe C++
ISTypeCPP	Verifica se o tipo é um tipo padrão C++
GetLisParCPP	Retorna a lista de parâmetros de um método C++

4.1.3 CLASSE TCLASSLAYOUT

Esta classe representa a estrutura de uma classe, seja ela extraída de um arquivo do Rose ou de um fonte C++. As propriedades e métodos estão ilustrados na figura 19.

Figura 19 – Classe TClassLayout.



As propriedades e métodos da classe TClassLayout são detalhados no quadro 4.

Quadro 4 – Propriedades/métodos da classe TClassLayout

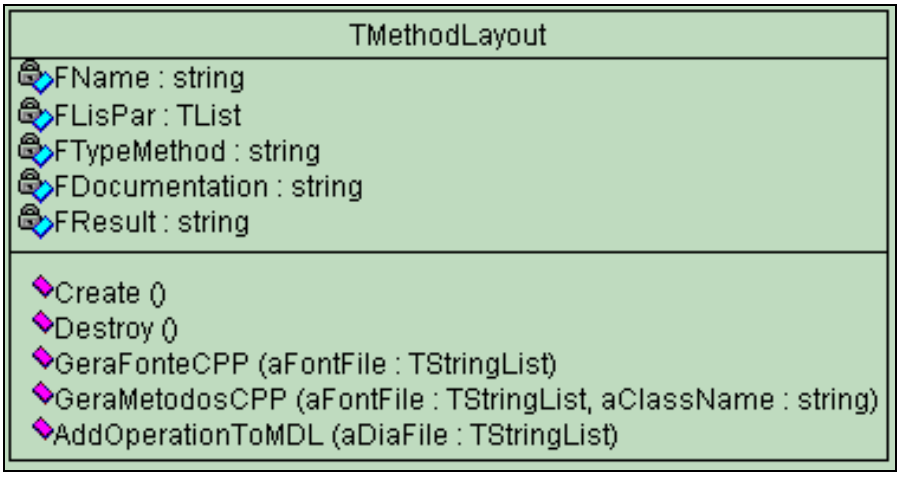
Propriedade/Método	Valor/Ação
FName	Nome da classe
FAtributes	Lista com os atributos da classe
FMethods	Lista com os métodos da classe
FSupClass	Contém o nome da classe pai
FDocumentation	Contém a documentação da classe
Create	Função construtora da classe

Destroy	Função destrutora da classe
GeraFonteCPP	Gera o fonte C++ da classe
GeraMetodosCPP	Gera a lista de métodos no fonte C++
GeraDiaRose	Gera o diagrama do Rose (definição da classe)

4.1.4 CLASSE TMETHODLAYOUT

Esta classe representa a estrutura de um método de uma classe. A figura 20 ilustra as propriedades e métodos da classe TMethodLayout.

Figura 20 – Classe TMethodLayout.



As propriedades e métodos da classe TMethodLayout estão detalhados no quadro 5.

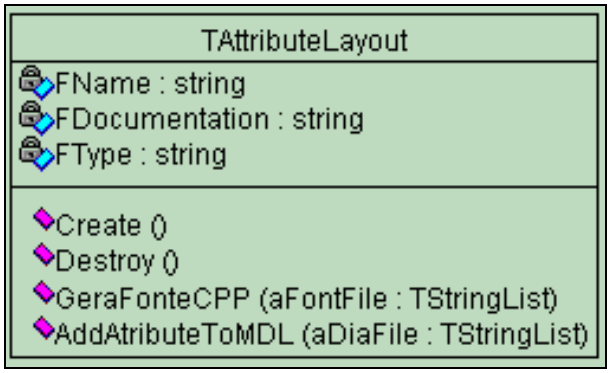
Quadro 5 – Propriedades/métodos da classe TMethodLayout

Propriedade/Método	Valor/Ação
FName	Nome do método.
FLisPar	Lista com os parâmetros do método
FTypeMethod	Tipo do método
FDocumentation	Contém a documentação da classe
FResult	Retorno da classe
Create	Função construtora da Classe
Destroy	Função destrutora da Classe
GeraFonteCPP	Gera o fonte C++ do método
GeraMetodosCPP	Gera o fonte C++ referente ao método
AddOperationToMDL	Gera a definição do método para o arquivo de diagramas do Rose

4.1.5 CLASSE TATTRIBUTELAYOUT

Esta classe representa a estrutura de um método de uma classe, tais como o nome do atributo, o tipo e a documentação. A figura 21 ilustra a classe TattributeLayout.

Figura 21 – Classe TAttributeLayout.



As propriedades e métodos da classe TAttributeLayout são detalhadas no quadro 6, abaixo:

Quadro 6 – Propriedades/métodos da classe TAttributeLayout

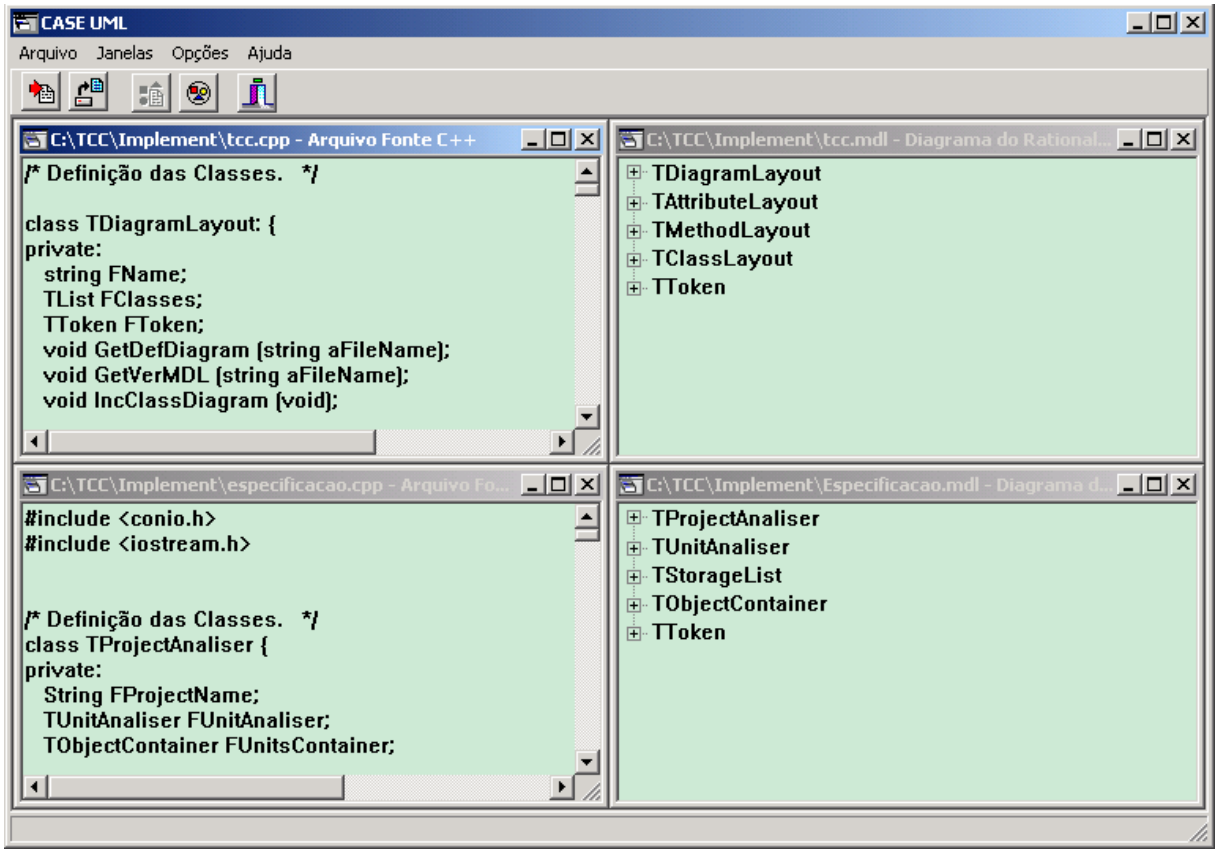
Propriedade/Método	Valor/Ação
Fname	Nome do atributo
FDocumentation	Contém a documentação do atributo
FType	Contém o tipo do atributo
Create	Função construtora da classe
Destroy	Função destrutora da Classe
GeraFonteCPP	Gera o fonte C++ do atributo
AddAtributeToMDL	Gera a definição do atributo para o arquivo de diagramas do Rose

4.2 FUNCIONAMENTO DO PROTÓTIPO

A seguir será descrito o funcionamento do protótipo, com a utilização do próprio diagrama de classes de classes utilizado na implementação do protótipo para a demonstração das suas funções.

A figura 22 mostra quatro janelas do protótipo organizadas através do item de menu Titulo, duas janelas com arquivos fonte C++ abertos e duas janelas com diagrama de classes aberto.

Figura 22 – Tela Principal do Protótipo Desenvolvido



O menu principal é composto pelos itens Arquivo, Janelas, Opções e Ajuda os quais se encontram na parte superior da janela principal do protótipo.

O principal diferencial do protótipo é a possibilidade de se utilizar uma ferramenta simples e de baixo custo, que possa ser útil no desenvolvimento de sistemas.

4.2.1 ITEM DE MENU ARQUIVO

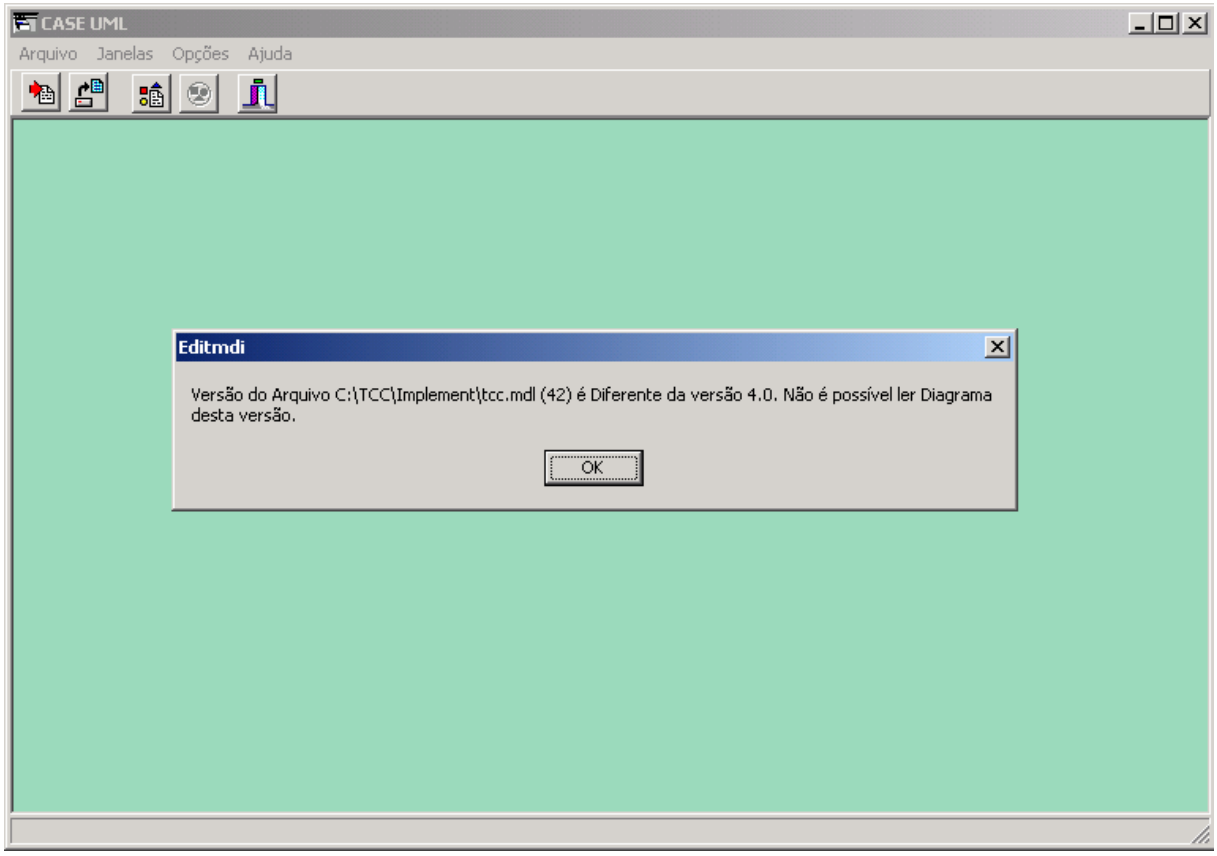
Este protótipo possui em sua janela principal no item de menu Arquivo as opções para Abrir Diagrama do Rose, Abrir Arquivo Fonte C++, Fechar a Janela Atual, Fechar Todas as Janelas e Sair.

Inicialmente para trabalhar-se com este protótipo é necessário abrir um arquivo Diagrama do Rose ou Arquivo Fonte C++. Este item de Menu pode ser acessado pela tecla de atalho ALT-A .

A opção Abrir Diagrama do Rose só abre arquivo da versão 4.0 do Rational Rose. Caso o arquivo que está sendo aberto não estiver na versão 4.0 o protótipo exibirá uma mensagem indicando versão incompatível do arquivo.

A figura 23 ilustra a mensagem de erro que o protótipo mostra, caso a versão do diagrama do Rose que o usuário está tentando abrir seja diferente da versão 4.0

Figura 23 – Mensagem de versão incompatível do Diagrama

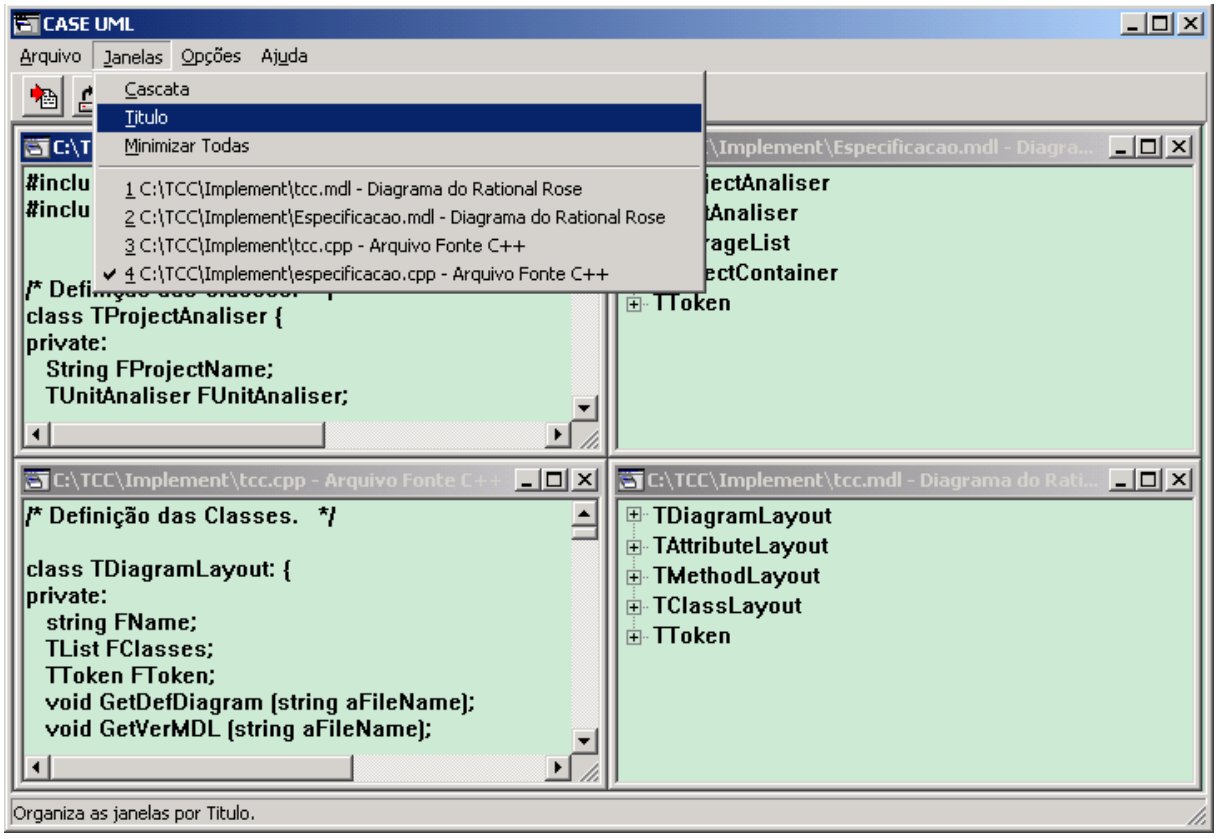


4.2.2 ITEM DE MENU JANELAS

No item de menu Janelas do menu principal existem várias opções para visualização das diversas janelas abertas pelo protótipo, bem como a lista de todas as janelas ativas no protótipo.

A figura 24 ilustra as opções do menu Janelas. Entre elas estão: Cascata, Titulo, Arranjar Ícones e Minimizar Todas.

Figura 24 – Item de Menu Janelas

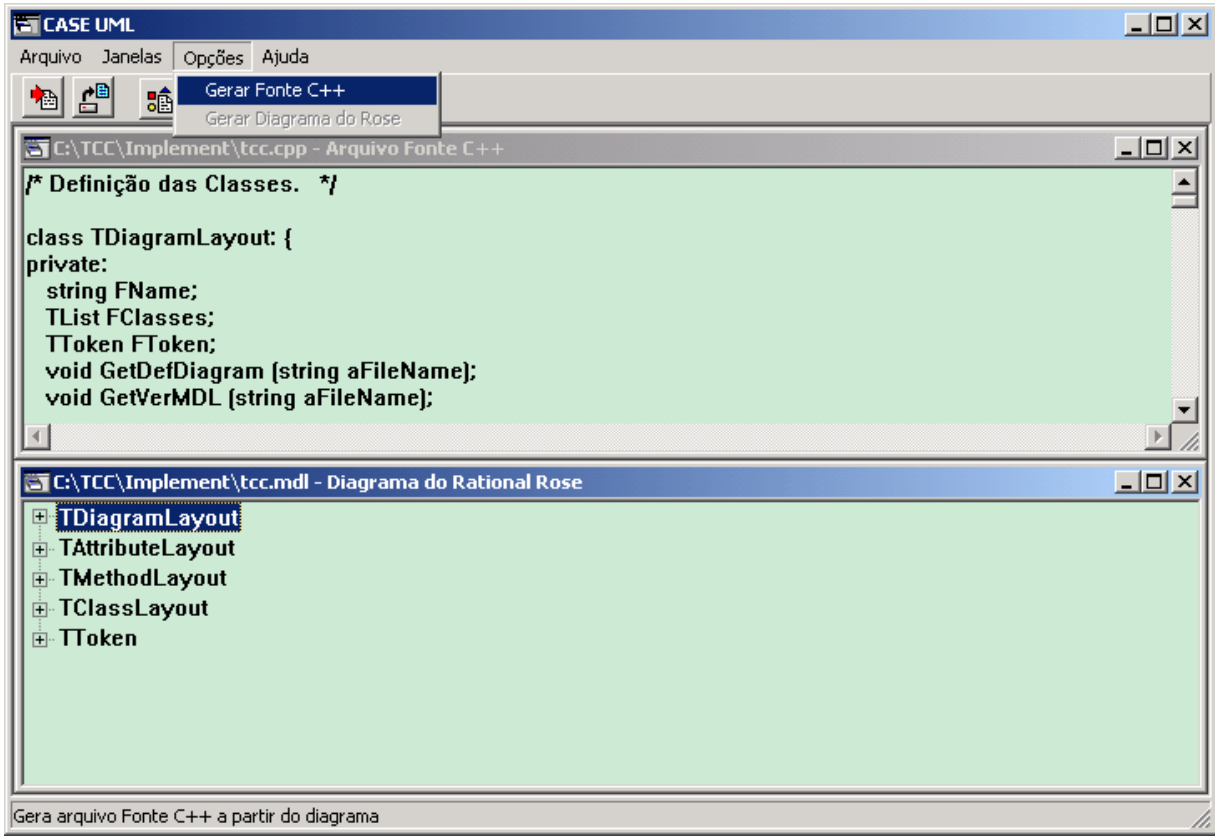


4.2.3 ITEM DE MENU OPÇÕES

Através do item de menu Opções é possível Gerar código fonte em C++ e também gerar o diagrama de classes para o Rational Rose 4.0 a partir de um código fonte C++. Conforme a janela ativa é habilitado a opção de Gerar Fonte C++ ou Gerar Diagrama do Rose.

A opção Gerar Fonte C++, gera um arquivo fonte C++ no padrão ANSI contendo a definição das classes e também todas as declarações dos métodos das classes. A figura 25 ilustra o item de menu Opções.

Figura 25 – Item de Menu Opções

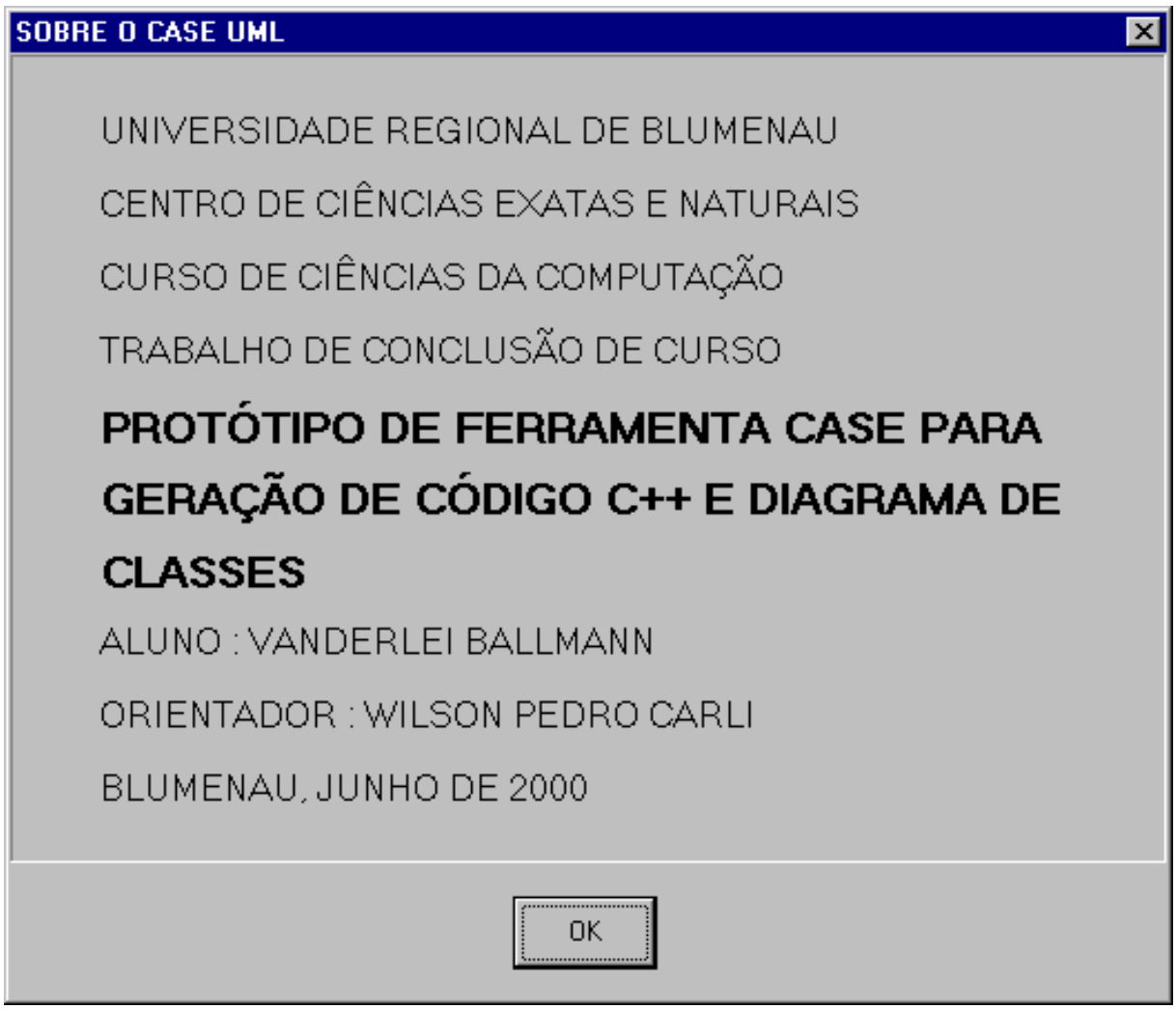


A opção Gerar Diagrama do Rose gera um arquivo .mdl contendo a definição das classes e suas ligações. Este arquivo poderá ser lido pela ferramenta Rational Rose 4.0.

4.2.4 ITEM DE MENU AJUDA

O item de menu Ajuda, Sobre traz informações gerais sobre o desenvolvimento do protótipo, conforme a figura 26.

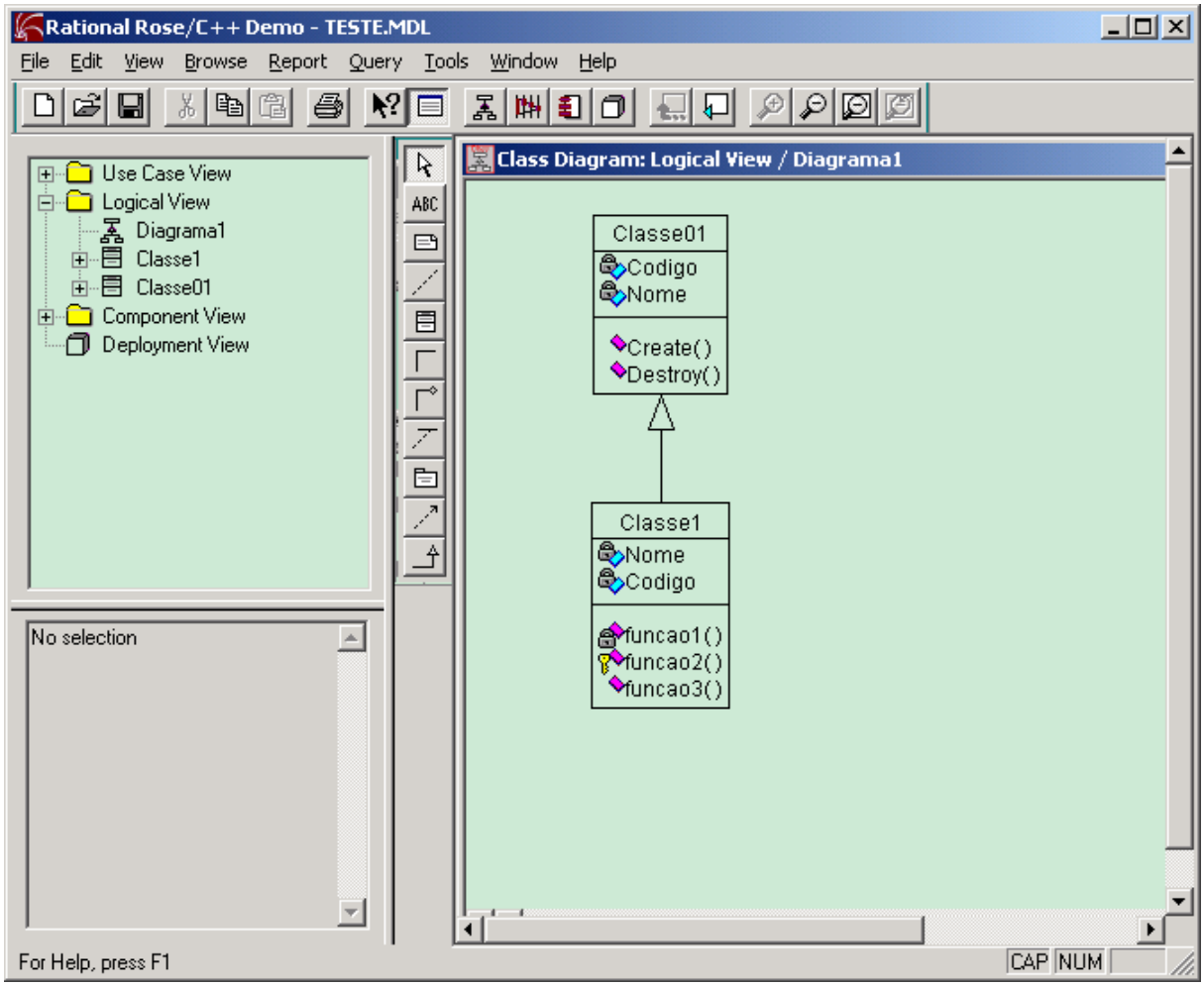
Figura 26 – Tela Sobre do Protótipo



4.2.5 EXEMPLO DE UTILIZAÇÃO DO PROTÓTIPO

A figura 27 ilustra um diagrama de classes hipotético, o qual será utilizado para demonstrar a geração de código C++.

Figura 27 – Diagrama exemplo para utilização do Protótipo



O quadro 7 ilustra o arquivo fonte C++ gerado pelo protótipo, a partir do diagrama de classes ilustrado na figura 27.

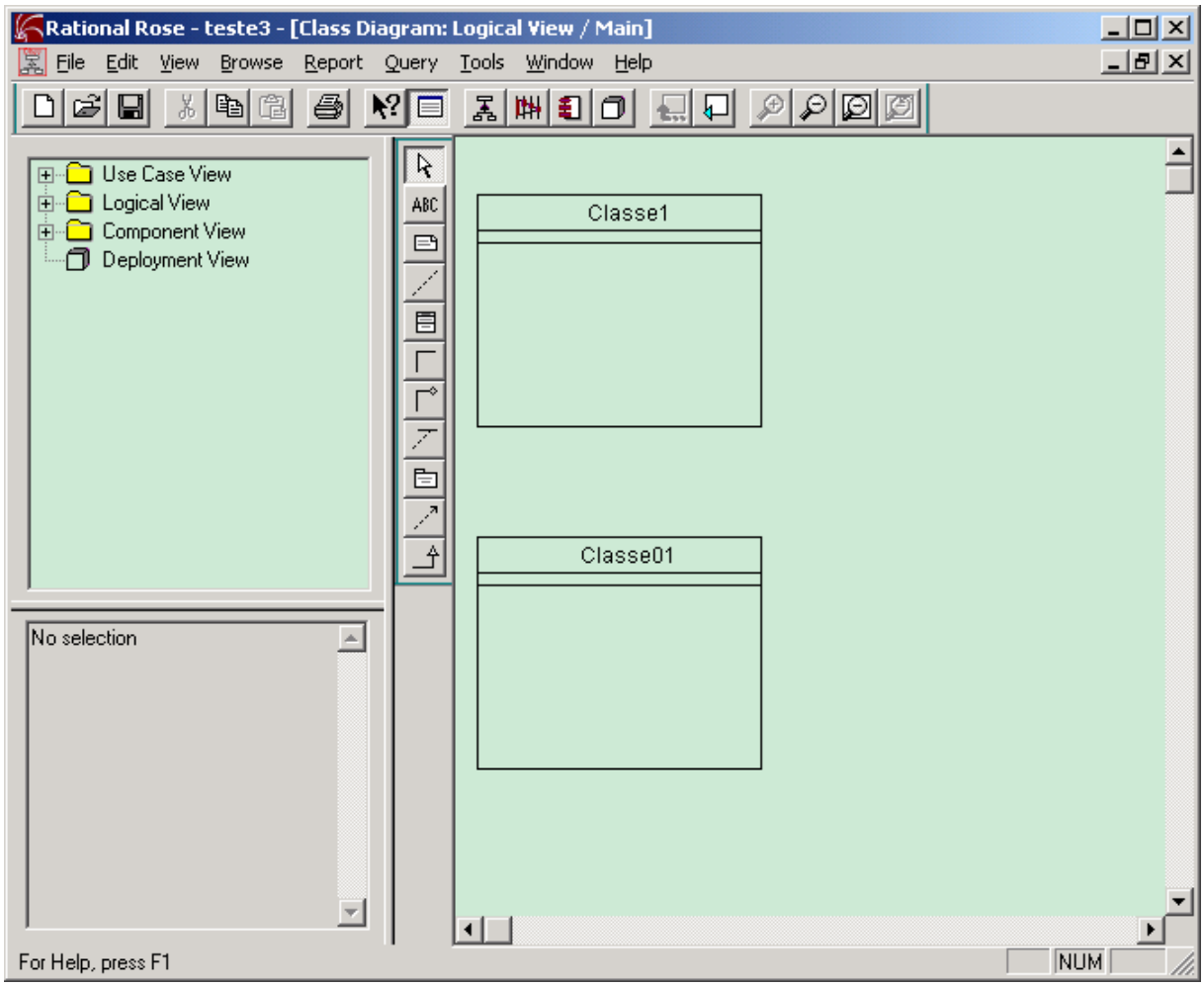
Quadro 7 - Exemplo de programa fonte C++ gerado pelo protótipo

```
/* Definição das Classes.   */
```

```
class Classe1: public Classe01 {
private:
   void *Nome;
   void *Codigo;
   void funcao1 (void);
protected:
   void funcao2 (void);
public:
   void funcao3 (void);
};

class Classe01 {
private:
   void *Codigo;
   void *Nome;
protected:
public:
   void Create (void);
   void Destroy (void);
};

/* Definição dos métodos das Classes.   */

void Classe1::funcao1 (void)
{

}

void Classe1::funcao2 (void)
{

}

void Classe1::funcao3 (void)
{

}

void Classe01::Create (void)
{

}

void Classe01::Destroy (void)
{

}
```

A figura 28 ilustra o diagrama de classes gerado pelo protótipo, a partir do arquivo fonte C++ ilustrado no quadro 7. O protótipo gera apenas a definição das classes com os seus atributos e métodos, os quais não são mostrados. O protótipo não gera as ligações entre as classes.

Figura 28 – Diagrama de Classes gerado pelo Protótipo



5 CONCLUSÕES E SUGESTÕES

Este capítulo apresenta as conclusões, dificuldades encontradas e sugestões referentes ao trabalho desenvolvido.

5.1 CONCLUSÕES

A tecnologia da orientação a objetos promove uma mudanças na maneira como os programas são implementados e na maneira como os analistas devem modelar o sistema antes mesmo da sua implementação em si. Com a orientação a objetos tem surgido métodos ou linguagens para modelagem de sistemas OO, entre elas destaca-se a UML. Com a UML, surgiram várias ferramentas CASE-OO que a suportam. Estas ferramentas contribuem para o sucesso de um software construído através da tecnologia da orientação a objetos.

Dentre as limitações deste protótipo, estão a possibilidade de leitura de diagramas somente da versão 4.0 da ferramenta Rational Rose e a análise de apenas um arquivo fonte em C++.

Com o desenvolvimento deste protótipo, que é na verdade, uma pequena parte do que uma ferramenta CASE-OO pode fazer, mostrou o grande auxílio que estas ferramentas podem dar na especificação e desenvolvimento de sistemas orientados a objeto.

Apesar de todos os benefícios da modelagem de sistemas, isto ainda é muito pouco utilizado pelas empresas de software. O protótipo criado poderia auxiliar no sentido de se criar documentação a partir de arquivo fonte em C++, pois a partir de diagramas é mais fácil analisar o sistema existente e até encontrar possíveis erros de projeto.

5.2 DIFICULDADES ENCONTRADAS

As maiores dificuldades encontradas na fase do levantamento bibliográfico, foram a pouca quantidade de material sobre UML na biblioteca, pois os livros em português sobre o assunto são muito concorridos.

Na parte da implementação, a maior dificuldade foi encontrada na rotina que gera a definição das classes para a ferramenta Rational Rose, quanto ao melhor posicionamento das classes e suas ligações dentro do diagrama. Pode-se dizer que esta dificuldade aumenta conforme o número de classes e o número de ligações entre estas classes, já que o número de possibilidades para o melhor posicionamento é muito grande.

5.3 SUGESTÕES

Sugere-se que seja aperfeiçoado a parte de engenharia reversa de código fonte, onde o protótipo lê o arquivo fonte em C++ e gera o diagrama de classes para o Rational Rose. Poderia ser implementada uma rotina mais aperfeiçoada para posicionar as classes da melhor forma possível no diagrama, considerando herança, associação e agregação.

Outra melhoria, poderia ser incluir também a geração de código fonte em Object Pascal. Desta forma, através da estrutura de classes criada, o protótipo poderia servir também para converter um fonte C++ para Object Pascal e vice-versa.

ANEXO I – FONTES DA CLASSE TATTRIBUTELAYOUT

```
{ Class TAttributeLayout }
constructor TAttributeLayout.Create;
begin
FName := '';
FType := '';
FDocumentation := '';
end;

{ Gera a string com a declaracao do atributo em C++. }
procedure TAttributeLayout.GeraFonteCPP (var aFontFile : TStringList);
begin
if   (Length (FType) = 0) then
    aFontFile.Add ('   void *' + FName + ';')
else aFontFile.Add ('   ' + FType + ' ' + FName + ';');
end;

{ Gera a definicao do atributo para o arquivo .MDL do Rose }
procedure TAttributeLayout.AddAttributeToMDL (aDiagFile : TStringList);
begin
aDiagFile.Add ('(object ClassAttribute "' + FName + '"');
aDiagFile.Add (' quid          "000000000000"');
aDiagFile.Add (' documentation "' + FDocumentation + '"');
aDiagFile.Add (' type          "' + FType + '")');
end;
```

ANEXO II – FONTES DA CLASSE TMETHODLAYOUT

```
constructor TMethodLayout.Create;
begin
FName := '';
FTypeMethod := '';
FDocumentation := '';
FLisPar := TList.Create;
end;

destructor TMethodLayout.Destroy;
var
  x : Integer;
  xAttribute : TAttributeLayout;
begin
for x := 0 to FLisPar.Count-1 do
   begin
   xAttribute := FLisPar.Items[x];
   xAttribute.Free;
   end;
FLisPar.Free;
end;

{ Gera a string com a declaracao do método em C++. }
procedure TMethodLayout.GeraFonteCPP (var aFontFile : TStringList);
var
  x : Integer;
  stratr : string;
  xatribute : TAttributeLayout;
begin
stratr := '';
if   (FLisPar.Count > 0) then
    begin
    for  x := 0 to (FLisPar.Count-1) do
        begin
        xatribute := FLisPar.Items[x];

        if   (x > 0) then
            stratr := stratr + ', ';

        stratr := stratr + xatribute.TypeAtr + ' ' + xatribute.FName;
        end;
    end
else stratr := 'void';

if   (length (FResult) = 0) then
    aFontFile.Add ('   void ' + FName + ' (' + stratr + ');')
else aFontFile.Add ('   ' + FResult + ' ' + FName + ' (' + stratr + ');');
end;

{ Gera o cabecalho da funcao em C++. }
procedure TMethodLayout.GeraMetodosCPP (var aFontFile : TStringList; aClassName : string);
var
```

```
  x : Integer;
  stratr : string;
  xatribute : TAttributeLayout;
begin
stratr := '';
if   (FLisPar.Count > 0) then
    begin
    for  x := 0 to (FLisPar.Count-1) do
        begin
        xatribute := FLisPar.Items[x];

        if   (x > 0) then
            stratr := stratr + ', ';

        stratr := stratr + xatribute.TypeAtr + ' ' + xatribute.FName;
        end;
    end
else stratr := 'void';

if   (length (FResult) = 0) then
    aFontFile.Add ('void ' + aClassName + '::' + FName + ' (' + stratr + ')')
else aFontFile.Add (FResult + ' ' + aClassName + '::'  + FName + ' (' + stratr + ')');

aFontFile.Add ('{');
aFontFile.Add (' ');
aFontFile.Add ('}');
end;

procedure TMethodLayout.AddOperationToMDL (aDiagFile : TStringList);
var
   x : Integer;
   xatribute : TAttributeLayout;
begin
aDiagFile.Add ('(object Operation "' + FName + '"');
aDiagFile.Add (' quid         "000000000000"');

if   (FLisPar.Count > 0) then
    begin
    aDiagFile.Add (' parameters       (list Parameters');

    for  x := 0 to (FLisPar.Count-1) do
        begin
        xatribute := FLisPar.Items[x];
        aDiagFile.Add (' (object Parameter "' + xatribute.Name + '"');
        aDiagFile.Add ('  type            "' + xatribute.TypeAtr + '"');
        if   (x = (FLisPar.Count-1)) then
            aDiagFile.Add ('  initv       "default"))')
        else aDiagFile.Add ('  initv       "default")');
        end;
    end;

aDiagFile.Add ('  result       "' + FResult + '"');
aDiagFile.Add ('  concurrency         "Sequential"');
aDiagFile.Add ('  opExportControl  "' + FTypeMethod + '"');
aDiagFile.Add ('  uid         0000000000)');
end;
```

ANEXO III – FONTES DA CLASSE TCLASSLAYOUT

```
{ Class TClassLayout }
constructor TClassLayout.Create;
begin
FAtributes := TList.Create;
FMethods := TList.Create;
FDocumentation := '';
end;

destructor TClassLayout.Destroy;
var
   x : Integer;
   xAttribute : TAttributeLayout;
   xMethod : TMethodLayout;
begin
for x := 0 to FAtributes.Count-1 do
    begin
    xAttribute := FAtributes.Items[x];
    xAttribute.Free;
    end;
FAtributes.Free;

for x := 0 to FMethods.Count-1 do
    begin
    xMethod := FMethods.Items[x];
    xMethod.Free;
    end;
FMethods.Free;
end;

{ Gera o cabecalho da Classe em C++. }
procedure TClassLayout.GeraFonteCPP (var aFontFile : TStringList);
var
   x : Integer;
   xAttribute : TAttributeLayout;
   xMethod : TMethodLayout;
begin
if   (Length (FSuperClass) > 0) then
     aFontFile.Add ('class ' + FName + ': public ' + FSuperClass + ' {')
else aFontFile.Add ('class ' + FName + ' {');
{ Gera Atributos e Metodos privados. }
aFontFile.Add ('private: ');
for x := 0 to FAtributes.Count-1 do
    begin
    xAttribute := FAtributes.Items[x];
    xAttribute.GeraFonteCPP (aFontFile);
    end;

for x := 0 to FMethods.Count-1 do
    begin
    xMethod := FMethods.Items[x];
    if   (UpperCase (xMethod.FTypeMethod) = 'PRIVATE') or (Length (xMethod.FTypeMethod) = 0) then
```

```
        xMethod.GeraFonteCPP (aFontFile);
    end;

{ Gera Atributos e Metodos protegidos. }
aFontFile.Add ('protected: ');

for x := 0 to FMethods.Count-1 do
    begin
    xMethod := FMethods.Items[x];
    if   (UpperCase (xMethod.FTypeMethod) = 'PROTECTED') then
        xMethod.GeraFonteCPP (aFontFile);
    end;

{ Gera Atributos e Metodos publicos. }
aFontFile.Add ('public: ');

for x := 0 to FMethods.Count-1 do
    begin
    xMethod := FMethods.Items[x];
    if   (UpperCase (xMethod.FTypeMethod) = 'PUBLIC') then
        xMethod.GeraFonteCPP (aFontFile);
    end;

aFontFile.Add ('};');
end;

{ Gera o cabecalho da Classe em C++. }
procedure TClassLayout.GeraMetodosCPP (var aFontFile : TStringList);
var
   x : Integer;
   xMethod : TMethodLayout;
begin

for x := 0 to FMethods.Count-1 do
    begin
    xMethod := FMethods.Items[x];
    xMethod.GeraMetodosCPP (aFontFile, FName);
    aFontFile.Add (' ');
    end;
end;


{ Gera a string com a definicao das classes para o Rose. }
procedure TClassLayout.GeraDefClasRose (aNumClas : Integer; var aDiagFile : TStringList);
var
   x : Integer;
   xquid  : string;
   xattribute : TAttributeLayout;
   xMethod : TMethodLayout;
begin

if   (aNumClas = 0) then
     aDiagFile.Add ('logical_models  (list unit_reference_list');

aDiagFile.Add ('(object Class "' + FName + '"');
xquid := Format ('%12.12d',[aNumClas+6]);
```

```
aDiagFile.Add (' quid         "' + xquid + '");


if   (FMethods.Count > 0) then
     begin
     aDiagFile.Add (' operations    (list Operations');

     for x := 0 to (FMethods.Count-1) do
        begin
        xMethod := FMethods.Items[x];
        xMethod.AddOperationToMDL (aDiagFile);
        end;

     aDiagFile.Add (' )');
     end;

if   (FAtributes.Count > 0) then
     begin
     aDiagFile.Add (' class_attributes             (list class_attribute_list');

     for x := 0 to FAtributes.Count-1 do
        begin
        xAttribute := FAtributes.Items[x];
        xAttribute.AddAttributeToMDL (aDiagFile);
        end;

     aDiagFile.Add (' )');
     end;

aDiagFile.Add (' )');
end;

{ Gera a string com a definicao das classes para o Rose (Posicionamento). }
procedure TClassLayout.GeraViewClasRose (aNumClas : Integer; var aViewFile : TStringList);
var
   xposclas, xposlab, xposcomp : Integer;
   xquid : string;
begin

if   (aNumClas = 0) then
     aViewFile.Add ('items               (list diagram_item_list');

xposclas := 304 + (aNumClas * 600);
xposlab := 110 + (aNumClas * 600);
xposcomp := 167 + (aNumClas * 600);

aViewFile.Add ('    (object ClassView "Class" "' + FName + '" @' + IntToStr (aNumClas+1));
//aViewFile.Add ('    IncludeAttribute          TRUE');
//aViewFile.Add ('    IncludeOperation          TRUE');
aViewFile.Add ('    IncludeAttribute          FALSE');
aViewFile.Add ('    IncludeOperation          FALSE');
aViewFile.Add ('    location       (288,' + IntToStr (xposclas) + ')');

{ Object ItemLabel }
aViewFile.Add ('    label                (object ItemLabel');
aViewFile.Add ('       Parent_View @' + IntToStr (aNumClas+1));
```

```
aViewFile.Add ('      location       (65,' + IntToStr (xposlab) + ')');
aViewFile.Add ('      max_width      479');
aViewFile.Add ('      justify        0');
aViewFile.Add ('      label          "' + FName + '")');

{ Object Compartment }
xquid := Format ('%12.12d',[aNumClas+6]);
aViewFile.Add ('    quidu            "' + xquid + '"');
aViewFile.Add ('    compartment      (object Compartment');
aViewFile.Add ('      Parent_View    @' + IntToStr (aNumClas+1));
aViewFile.Add ('      location       (186,' + IntToStr (xposcomp) + ')');
aViewFile.Add ('      anchor         2');
aViewFile.Add ('      nlines         7');
aViewFile.Add ('      max_width      500)');

aViewFile.Add ('    width            500');
aViewFile.Add ('    height           410');

aViewFile.Add ('    annotation       8');
aViewFile.Add ('    autoResize       TRUE)');
end;
```

ANEXO IV – FONTES DA CLASSE TDIAGRAMLAYOUT

```
constructor TDiagramLayout.Create;
begin
inherited Create;
FClasses := TList.Create;
FName := '';
FToken := TToken.Create;
end;

destructor TDiagramLayout.Destroy;
var
  x : Integer;
  xClasses : TClassLayout;
begin
for x := 0 to FClasses.Count-1 do
   begin
   xClasses := FClasses.Items[x];
   xClasses.Free;
   end;
FClasses.Free;
FToken.Free;
end;

{ Carrega Diagrama de Classes. }
procedure TDiagramLayout.LoadDiagram (aFileName : string);
begin
{ Inicializa Valores para GetToken. }
FToken.InitValTok (aFileName);

{ Verifica a versão do arquivo de Diagrama do Rose. }
GetVerMDL (aFileName);
{ Inicializa Valores para GetToken. }
FToken.InitValTok (aFileName);
FToken.GetToken;

while (FToken.TypeToken <> ttEnd) do
     begin
     { Verifica o Token 'Class'. }
     if   (FToken.Token = 'ClassDiagram') then
          begin
          IncClassDiagram;
          continue;
          end;

     { Verifica o Token 'Class'. }
     if   (FToken.Token = 'Class') then
          begin
          IncClass;
          continue;
          end;
```

```
    FToken.GetToken;
    end;
end;

{ Gera Fonte em C++ das classes. }
procedure TDiagramLayout.GeraFonteCPP (aFileName : string);
var
  x : Integer;
  xFontFile : TStringList;
  xClasses : TClassLayout;
begin

xFontFile := TStringList.Create;
try
    { Gera a estrutura das Classes no C++. }
    if   (FClasses.Count > 0) then
        GeraCabecalhoCPP (xFontFile);

    for  x := 0 to (FClasses.Count-1) do
        begin
        xClasses := FClasses.Items[x];
        xClasses.GeraFonteCPP (xFontFile);
        xFontFile.Add (' ');
        end;

    if   (FClasses.Count > 0) then
        begin
        xFontFile.Add (' ');
        xFontFile.Add (' ');
        xFontFile.Add ('/* Definição dos métodos das Classes.   */');
        xFontFile.Add (' ');
        end;

    { Gera a definição das funções no C++. }
    for  x := 0 to (FClasses.Count-1) do
        begin
        xClasses := FClasses.Items[x];
        xClasses.GeraMetodosCPP (xFontFile);
        xFontFile.Add (' ');
        end;

    if   (FClasses.Count > 0) then
        begin
        xFontFile.Add ('/* Fim do Arquivo Fonte.   */');
        xFontFile.SaveToFile (aFileName);
        end;
finally
    xFontFile.Free;
end;

MessageDlg ('Arquivo Fonte ' + aFileName + ' Gerado.', mtInformation, [mbOk], 0);
end;


{ Gera Cabecalho do Arquivo Fonte em C++. }
```

```
procedure TDiagramLayout.GeraCabecalhoCPP (var aFontFile : TStringList);
begin
aFontFile.Add ('/* Definição das Classes.   */');
aFontFile.Add (' ');
end;
{ Verifica a versão do arquivo de Diagrama do Rose. }
procedure TDiagramLayout.GetVerMDL (aFileName : string);
var
   version : Integer;
   strmsg : string;
begin
FToken.GetToken;
version := 0;

while (FToken.TypeToken <> ttEnd) do
     begin
     { Verifica a versão do arquivo .mdl. }
     if   (FToken.Token = 'Petal') then
          begin
          FToken.GetToken;
          if   (FToken.Token = 'version') then
               begin
               FToken.GetToken;
               if   (FToken.TypeToken = ttNumber) then
                    version := StrToInt (FToken.Token);
               end;

          break;
          end;

    FToken.GetToken;
    end;

if   (version = 0) then
    Raise Exception.Create ('Versão do Arquivo não encontrada. Formato do Arquivo inválido.')
else begin
    if   (version <> 40) then
         begin
         strmsg := Format ('Versão do Arquivo %s (%d) é Diferente da versão 4.0. Não é possível ler Diagrama
desta versão.',[aFileName,version]);
         Raise Exception.Create (strmsg);
         end;
    end;
end;


procedure TDiagramLayout.IncClassDiagram;
begin
FToken.GetToken;

if   (FToken.Token = '"') and (FToken.TypeToken = ttsymbol) then
    begin
    FToken.GetToken;
    if   (FToken.TypeToken = ttIdentify) then
         FName := FToken.Token;
    end;
```

```
end;



procedure TDiagramLayout.IncClass;
var
  qtdpar : Integer;
  xclass : TClassLayout;
  xattribute : TAttributeLayout;
  xmethod : TMethodLayout;
begin
FToken.GetToken;

if   (FToken.Token = "") and (FToken.TypeToken = ttsymbol) then
    begin
    FToken.GetToken;
    if   (FToken.TypeToken = ttIdentify) then
        begin
        xclass := TClassLayout.Create;
        xclass.Name := FToken.Token;
        end
    else exit;
    end
else exit;

FToken.GetToken;
qtdpar := 1;

while (FToken.TypeToken <> ttEnd) do
     begin
     if   (FToken.Token = 'Class') or (qtdpar = 0) then
          break;

     if   (FToken.Token = '(') then
          Inc (qtdpar)
     else
     if   (FToken.Token = ')') then
          Dec (qtdpar)
     else
     if   (FToken.Token = 'Operation') then
          begin
          xmethod := IncMethod;
          if   (xmethod <> nil) then
               xclass.FMethods.Add (xmethod);
          continue;
          end
     else
     if   (FToken.Token = 'ClassAttribute') then
          begin
          xattribute := IncAtribute;
          if   (xattribute <> nil) then
               xclass.FAtributes.Add (xattribute);
          continue;
          end
     else
```

```
    if   (FToken.Token = 'superclasses') then
         begin
         xclass.FSuperClass := GetSupClass;
         continue;
         end
    else
    if   (FToken.Token = 'documentation') then
         begin
         FToken.GetToken;
         if   (FToken.Token = "") and (FToken.TypeToken = ttsymbol) then
              begin
              FToken.GetToken;
              if   (FToken.TypeToken = ttIdentify) then
                   xclass.FDocumentation := FToken.Token;
              end;
         end;

    FToken.GetToken;
    end;

{ Caso xclass está vazio ou a classe não possui nenhum método ou atributo, Não inclui na lista. }
if   (xclass <> nil) and ((xclass.Atributes.Count > 0) or (xclass.Methods.Count > 0)) then
    FClasses.Add (xclass);
end;

function TDiagramLayout.IncMethod : TMethodLayout;
var
  qtdpar : Integer;
  xmethod : TMethodLayout;
begin
Result := nil;
FToken.GetToken;

if   (FToken.Token = "") and (FToken.TypeToken = ttsymbol) then
    begin
    FToken.GetToken;
    if   (FToken.TypeToken = ttIdentify) then
         begin
         xmethod := TMethodLayout.Create;
         xmethod.Name := FToken.Token;
         end
    else exit;
    end
else exit;

FToken.GetToken;
qtdpar := 1;

while (FToken.TypeToken <> ttEnd) do
     begin
     if    (FToken.Token = 'superclasses') or (FToken.Token = 'operations') or (FToken.Token = 'ClassAttribute')
or (qtdpar = 0) then
          break;

     if   (FToken.Token = '(') then
          Inc (qtdpar)
```

```
    else
    if   (FToken.Token = ')') then
        Dec (qtdpar)
    else
    if   (FToken.Token = 'opExportControl') then
        begin
        FToken.GetToken;
        if   (FToken.Token = '"') and (FToken.TypeToken = ttsymbol) then
            begin
            FToken.GetToken;
            if   (FToken.TypeToken = ttIdentify) then
                xmethod.TypeMethod := FToken.Token;
            end;
        end
    else
    if   (FToken.Token = 'result') then
        begin
        FToken.GetToken;
        if   (FToken.Token = '"') and (FToken.TypeToken = ttsymbol) then
            begin
            FToken.GetToken;
            if   (FToken.TypeToken = ttIdentify) then
                xmethod.FResult := FToken.Token;
            end;
        end
    else
    if   (FToken.Token = 'parameters') then
        begin
        GetListParameters (xmethod);
        continue;
        end;

    FToken.GetToken;
    end;

Result := xmethod;
end;

procedure TDiagramLayout.GetListParameters (var aMethod : TMethodLayout);
var
  qtdpar : Integer;
  xattribute : TAttributeLayout;
begin
FToken.GetToken;
qtdpar := 1;
if   (FToken.Token = '(') then
    FToken.GetToken
else exit;

while (FToken.TypeToken <> ttEnd) do
    begin
    if    (FToken.Token = 'superclasses') or (FToken.Token = 'operations') or (FToken.Token = 'ClassAttribute')
or (qtdpar = 0) then
        break;

    if   (FToken.Token = '(') then
```

```
        Inc (qtdpar)
    else
    if   (FToken.Token = ')') then
        Dec (qtdpar)
    else
    if   (FToken.Token = 'Parameter') then
        begin
        xattribute := IncAtribute;
        if   (xattribute <> nil) then
            aMethod.FLisPar.Add (xattribute);

        Dec (qtdpar);
        continue;
        end;

    FToken.GetToken;
    end;
end;

function TDiagramLayout.IncAtribute : TAttributeLayout;
var
  qtdpar : Integer;
  xattribute : TAttributeLayout;
begin
Result := nil;
FToken.GetToken;

if   (FToken.Token = '"') and (FToken.TypeToken = ttsymbol) then
    begin
    FToken.GetToken;
    if   (FToken.TypeToken = ttIdentify) then
        begin
        xattribute := TAttributeLayout.Create;
        xattribute.Name := FToken.Token;
        end
    else exit;
    end
else exit;

FToken.GetToken;
qtdpar := 1;

while (FToken.TypeToken <> ttEnd) do
    begin
    if    (FToken.Token = 'superclasses') or (FToken.Token = 'operations') or (FToken.Token = 'ClassAttribute')
or (qtdpar = 0) then
        break;

    if   (FToken.Token = '(') then
        Inc (qtdpar)
    else
    if   (FToken.Token = ')') then
        Dec (qtdpar)
    else
    if   (FToken.Token = 'type') then
        begin
```

```
        FToken.GetToken;
        if   (FToken.Token = "") and (FToken.TypeToken = ttsymbol) then
            begin
            FToken.GetToken;
            if   (FToken.TypeToken = ttIdentify) then
                xattribute.FType := FToken.Token;
            end;
        end
    else
    if   (FToken.Token = 'documentation') then
        begin
        FToken.GetToken;
        if   (FToken.Token = "") and (FToken.TypeToken = ttsymbol) then
            begin
            FToken.GetToken;
            if   (FToken.TypeToken = ttIdentify) then
                xattribute.FDocumentation := FToken.Token;
            end;
        end;

    FToken.GetToken;
    end;

Result := xattribute;
end;

function TDiagramLayout.GetSupClass : string;
var
  qtdpar : Integer;
begin
Result := '';
qtdpar := 1;
FToken.GetToken;
if   (FToken.Token = '(') and (FToken.TypeToken = ttsymbol) then
    FToken.GetToken
else exit;

while (FToken.TypeToken <> ttEnd) do
    begin
    if    (FToken.Token = 'superclasses') or (FToken.Token = 'operations') or (FToken.Token = 'ClassAttribute')
or (qtdpar = 0) then
        break;

    if   (FToken.Token = '(') then
        Inc (qtdpar)
    else
    if   (FToken.Token = ')') then
        Dec (qtdpar)
    else
    if   (FToken.Token = 'supplier') then
        begin
        FToken.GetToken;
        if   (FToken.Token = "") and (FToken.TypeToken = ttsymbol) then
            begin
            FToken.GetToken;
            if   (FToken.TypeToken = ttIdentify) then
```

```
                Result := FToken.Token;
              end;
          continue;
          end;

      FToken.GetToken;
      end;
end;

{ Carrega o Arquivo Fonte C++ para Diagrama de Classes. }
procedure TDiagramLayout.LoadFontCPP (aFileName : string);
var
  xclass : TClassLayout;
begin
{ Inicializa Valores para FToken.GetToken. }
FToken.InitValTok (aFileName);

FToken.GetToken;

while (FToken.TypeToken <> ttEnd) do
     begin
     { Verifica o FToken.Token 'Class'. }
     if   (FToken.Token = 'class') then
          begin
          xclass := GetClassFromCPP;
          if   (xclass <> nil) then
               FClasses.Add (xclass);
          continue;
          end;

     { Verifica se terminou a parte de Definição das Classes. }
     if   (FToken.Token = ':') then
          begin
          FToken.GetToken;
          if   (FToken.Token = ':') then
               break
          else continue;
          end;

    FToken.GetToken;
    end;
end;

function TDiagramLayout.GetClassFromCPP : TClassLayout;
var
  xtype : string;
  xname : string;
  xdatatype : string;
  xclass : TClassLayout;
  xattribute : TAttributeLayout;
  xmethod : TMethodLayout;
begin
FToken.GetToken;
if   (FToken.TypeToken = ttIdentify) then
     begin
     xclass := TClassLayout.Create;
```

```
    xclass.Name := FToken.Token;
    end
else Raise Exception.Create ('Esperado : Nome da Classe');


{ Busca Hierarquia da Classe. }
xclass.FSuperClass := GetSupClassFromCPP;
if   (FToken.Token <> '{') then
    Raise Exception.Create ('Esperado : {');

FToken.GetToken;
if   (Pos (FToken.Token, 'public,protected,private') > 0) then
    begin
    xtype := FToken.Token;
    FToken.GetToken;
    if   (FToken.Token = ':') then
        FToken.GetToken;
    end
else xtype := 'private';

while (FToken.TypeToken <> ttEnd) do
     begin
     if   (FToken.Token = '}') then
          break;

     if   (Pos (FToken.Token, 'public,protected,private') > 0) then
          begin
          xtype := FToken.Token;
          FToken.GetToken;
          if   (FToken.Token = ':') then
               FToken.GetToken;
          continue;
          end;

     xdatatype := GetDataTypeFromCPP;

     if   (FToken.TypeToken = ttIdentify) then
          begin
          xname := FToken.Token;
          FToken.GetToken;
          if   (FToken.Token = '(') then
               begin
               if   (Length (xdatatype) = 0) then
                    Raise Exception.Create ('Esperado : Tipo do Método : ' + xname);

               xmethod := TMethodLayout.Create;
               xmethod.FName := xname;
               xmethod.FTypeMethod := xtype;
               xmethod.FResult := xdatatype;
               { Retorna a Lista de Parametros da funcao. }
               xmethod.FLisPar := GetLisParCPP;
               xclass.FMethods.Add (xmethod);
               end
          else
          if   (FToken.Token = ';') then
               begin
```

```
            if   (Length (xdatatype) = 0) then
                Raise Exception.Create ('Esperado : Tipo do Atributo : ' + xname);

            xattribute := TAttributeLayout.Create;
            xattribute.FName := xname;
            xattribute.FType := xdatatype;
            xclass.FAtributes.Add (xattribute);
            end;
        end
    else begin
        xclass.Free;
        Raise Exception.Create ('Esperado : Nome do Método ou Atributo');
        end;

    FToken.GetToken;
    end;

Result := xclass;
end;

{ Busca Hierarquia da Classe. }
function TDiagramLayout.GetSupClassFromCPP : string;
begin
FToken.GetToken;
if   (FToken.Token = ':') then
    begin
    FToken.GetToken;
    if   (Pos (FToken.Token, 'public,protected,private') > 0) then
        FToken.GetToken;

    while (FToken.TypeToken <> ttEnd) do
        begin
        if   (FToken.Token = '{') then
            break;

        Result := Result + FToken.Token;

        FToken.GetToken;
        end;
    end
else
if   (FToken.Token = '{') then
    exit
else Raise Exception.Create ('Esperado : {');
end;

function TDiagramLayout.GetDataTypeFromCPP : string;
var
    xtokant : string;
begin
Result := '';
xtokant := FToken.Token;

while (FToken.TypeToken <> ttEnd) do
    begin
    if   (FToken.Token = '}') then
```

```
          break;

      xtokant := FToken.Token;
      FToken.SavePos (FToken.Token);
      FToken.GetToken;
      if   (FToken.Token = '(') or (FToken.Token = ';') then
          begin
          FToken.RestorePos;
          break;
          end;

      Result := Result + ' ' + xtokant;
      end;
end;

{ Retorna a Lista de Parametros da funcao. }
function TDiagramLayout.GetLisParCPP : TList;
var
   xlispar : TList;
   xtoken : string;
   xattribute : TAttributeLayout;
begin
xlispar := TList.Create;

xattribute := nil;

FToken.GetToken;

while (FToken.TypeToken <> ttEnd) do
     begin
     if   (FToken.Token = '}') then
          break;

     if   (FToken.TypeToken = ttIdentify) and (xattribute = nil) then
          begin
          xattribute := TAttributeLayout.Create;
          xattribute.TypeAtr := FToken.Token;
          end
     else begin
          FToken.GetToken;
          continue;
          end;

     FToken.GetToken;

     if   (FToken.TypeToken = ttIdentify) then
          begin
          xattribute.FName := FToken.Token;
          FToken.GetToken;
          end;

     if   (FToken.Token = ';') or (FToken.Token = ',') or (FToken.Token = ')') then
          begin
          xlispar.Add (xattribute);
          xattribute := nil;
          xtoken := FToken.Token;
```

```
        FToken.GetToken;
        if   (xtoken = ',') then
             continue
        else break;
        end;
    end;

Result := xlispar;
end;


{ Gera Diagrama a partir do arquivo Fonte Carregado. }
procedure TDiagramLayout.GeraDiagrama (aFileName : string);
var
  x : Integer;
  xDefClass : TStringList;
  xViewClass : TStringList;
  xOutFile : TStringList;
  xclass : TClassLayout;
begin
if   (FClasses.Count = 0) then
    Raise Exception.Create ('Nenhuma Classe carregada do arquivo Fonte C++.');

try
    xOutFile := TStringList.Create;
    xDefClass := TStringList.Create;
    xViewClass := TStringList.Create;

    { Gera o Cabecalho do Arquivo MDL }
    GeraCabecalhoMDL (xOutFile, aFileName);

    { Gera a definição das classes e atributos para o Rose (MDL). }
    for  x := 0 to (FClasses.Count-1) do
        begin
        xClass := FClasses.Items[x];
        xClass.GeraDefClasRose (x, xDefClass);
        xClass.GeraViewClasRose (x, xViewClass);
        end;

    xViewClass.Add ('))))');
    xDefClass.Add (')');

    xOutFile.Add (xDefClass.Text);

    GeraLogPres (xOutFile);

    xOutFile.Add (xViewClass.Text);
    xOutFile.Add (')');

    xOutFile.SaveToFile (aFileName);
finally
    xOutFile.Free;
    xDefClass.Free;
    xViewClass.Free;
end;
```

```
MessageDlg ('Diagrama do Rational Rose (versão 4.0): ' + aFileName + ' Gerado.', mtInformation, [mbOk], 0);

end;

{ Gera o Cabecalho do Arquivo MDL }
procedure TDiagramLayout.GeraCabecalhoMDL (var aOutFile : TStringList; aFileName : string);
begin
aOutFile.Add ('(object Petal');
aOutFile.Add ('    version             40)');
aOutFile.Add ('(object Design "Logical View"');
aOutFile.Add ('    is_unit             TRUE');
aOutFile.Add ('    is_loaded           TRUE');
aOutFile.Add ('    file_name           "' + aFileName + '"); // (Ver)
aOutFile.Add ('    quid       "000000000001"');
aOutFile.Add ('    defaults            (object defaults');
aOutFile.Add ('    rightMargin         0.250000');
aOutFile.Add ('    leftMargin          0.250000');
aOutFile.Add ('    topMargin           0.250000');
aOutFile.Add ('                bottomMargin     0.500000');
aOutFile.Add ('                pageOverlap      0.250000');
aOutFile.Add ('                clipIconLabels   TRUE');
aOutFile.Add ('                autoResize       TRUE');
aOutFile.Add ('    snapToGrid          TRUE');
aOutFile.Add ('    gridX               16');
aOutFile.Add ('    gridY               16');
aOutFile.Add ('    defaultFont         (object Font');
aOutFile.Add ('                    size            9');
aOutFile.Add ('                    face            "helvetica"');
aOutFile.Add ('                    bold            FALSE');
aOutFile.Add ('                    italics         FALSE');
aOutFile.Add ('        underline       FALSE');
aOutFile.Add ('        strike          FALSE');
aOutFile.Add ('        color           0');
aOutFile.Add ('        default_color   TRUE)');
aOutFile.Add ('                showMessageNum          1');
aOutFile.Add ('                showClassOfObject       TRUE');
aOutFile.Add ('                notation            "Unified")');
aOutFile.Add ('        root_usecase_package    (object Class_Category "Use Case View"');
aOutFile.Add ('    quid       "000000000002"');
aOutFile.Add ('    exportControl       "Public"');
aOutFile.Add ('    global              TRUE');
aOutFile.Add ('    logical_models      (list unit_reference_list)');
aOutFile.Add ('                logical_presentations       (list unit_reference_list');
aOutFile.Add ('                    (object UseCaseDiagram "Main"');
aOutFile.Add ('                        quid        "000000000003"');
aOutFile.Add ('                        title       "Main"');
aOutFile.Add ('                zoom                100');
aOutFile.Add ('                max_height          28350');
aOutFile.Add ('                max_width           21600');
aOutFile.Add ('                origin_x            0');
aOutFile.Add ('                        origin_y            0');
aOutFile.Add ('                        items               (list diagram_item_list))))');
aOutFile.Add ('        root_category   (object Class_Category "Logical View"');
aOutFile.Add ('                quid        "000000000004"');
aOutFile.Add ('    exportControl       "Public"');
```

```
aOutFile.Add ('    global               TRUE');
aOutFile.Add ('    subsystem            "Component View"');
aOutFile.Add ('    quidu                "000000000005"');
end;

{ Gera o Cabecalho do logical presentations (Visualização das Classes) }
procedure TDiagramLayout.GeraLogPres (var aOutFile : TStringList);
begin
aOutFile.Add ('     logical_presentations       (list unit_reference_list');
aOutFile.Add ('         (object ClassDiagram "Main"');
aOutFile.Add ('                 quid        "3934312D0296"');
aOutFile.Add ('                 title       "Main"');
aOutFile.Add ('                 zoom                100');
aOutFile.Add ('                 max_height          28350');
aOutFile.Add ('                 max_width           21600');
aOutFile.Add ('                 origin_x            0');
aOutFile.Add ('                 origin_y            0');
end;
```

REFERÊNCIAS BIBLIOGRÁFICAS

[BAR1996] BARBIERI, Carlos. **Receita de objetos.** Artigo Computerworld, pp 18-19 : 20/05/1996.

[COA1991] COAD, Peter, YOURDON, Ed. **Análise baseada em objetos.** Rio de Janeiro : Campus, 1991.

[COL1994] COLEMAN, Derek; ARNOLD, Patrick; BODOFF, Stephanie; et all. **Object-oriented development – the fusion method**. New Jersey, EUA : Prentice Hall International Editions, 1994.

[ERI1998] ERIKSSON, Hans-Erik; PENKER, Magnus. **UML Toolkit**. New York, EUA : John Wiley & Sons, Inc.

[FUG1993] FUGGETA, Alfonso. **A classification of CASE Tecnology: Computer.** IEEE Computer Society, December, 1993.

[FUR1994] FURLAN, JOSÉ DAVI. **Reengenharia da informação – do mito à realidade .** São Paulo : Makron Books, 1994.

[FUR1998] FURLAN, JOSÉ DAVI. **Modelagem de Objetos através da UML.** São Paulo : Makron Books, 1998.

[GAN1990] GANE, Chris. **CASE: o relatório Gane**. Rio de Janeiro : Livros Técnicos e Científicos, 1990.

[GRA1994] GRAHAM, Ian. **Migrating to object technology**.California, EUA : Addison-Wesley, 1994.

[HAR1998] HARMON, Paul; WATSON, Mark. **Understanding UML – the developer´s guide**. San Francisco, California, EUA : Morgan Kaufmann Publishers, 1998.

[HOL1993] HOLZNER, Steven. **Programando em C++**. Rio de Janeiro : Campus, 1993.

[JOA1993] JOÃO, Belmiro dos Santos. **Metodologias de desenvolvimento de sistemas.** São Paulo : Editora Érica, 1993.

[MAR1994] MARTIN, James. **Princípios de análise e projeto baseados em objetos**. Rio de Janeiro : Campus, 1994.

[MCC1993] McCLURE, Carma. **The Three Rs of Software Automation: Re-Enginnering, Repository, Reusability**. New Jersey : Prentice Hall, 1993.

[PAB2000] PABLO, F. Rego Barros. **Linguagem de Modelagem Unificada, 2000.** Endereço Eletrônico: http://cc.usu.edu/~slqz9/uml.

[RAT1997] RATIONAL, Software Corporation. **Unified Modeling Language, version 1.1 1997.** Endereço Eletrônico: http://www/rational.com/uml.

[REE1995] REENSKAUG, Trygve. **Working with objects**. Massachusetts, EUA : Manning Publications Co, 1995.

[RUM1994] RUMBAUGH, James. **Modelagem e projetos baseados em objetos**. Rio de Janeiro : Campus, 1994.