UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

Protótipo de Gerador de Código Fonte baseado em Diagramas de Seqüências

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

LODEMAR JOSÉ HAFEMANN

BLUMENAU, JULHO/2000

2000/1-39

Protótipo de Gerador de Código Fonte baseado em Diagramas de Seqüências

LODEMAR JOSÉ HAFEMANN

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 P. Carli — Orientador na FURB

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

Prof. Wilson P. Carli

Prof. Everaldo Artur Grahl

Prof. Marcel Hugo

AGRADECIMENTOS

A Deus, pela minha vida e Sua presença constante, por nunca ter me abandonado nos momentos mais difíceis, nas horas de desânimo, por ter-me permitido chegar até aqui, agradeço pela benção.

À minha esposa Maria Cristina e à minha filha Layane, que sempre me incentivaram a ir avante. O amor, a disposição e os estímulos recebidos foram as forças necessárias para este triunfo. A vocês dedico esta conquista.

À Furb, a todo o corpo docente e demais funcionários, que no decorrer destes anos, nos ministraram seus conhecimentos e informações.

Ao Professor Orientador, Sr. Wilson P. Carli, e ao Coordenador do TCC, Sr. José Roque Voltolini da Silva, pelo período dedicado ao acompanhamento do TCC, orientação e avaliação dos trabalhos efetuados.

Aos meus pais, Luiz Hafemann (*in memorian*) e Martha T. Hafemann, que sem eles nada disso poderia ter se realizado. Ao meu irmão Nilton Hafemann que, mesmo de longe, sempre incentivou e acreditou na minha vitória.

Aos colegas e principalmente aos amigos angariados neste tempo de faculdade, entre eles Juan Carlo Krüger, André Luís Weber, Rubens Bósio, Marcelo Assis Costa, Clóvis Fabiano de Souza, Marcos Mueller, grandes amigos, que souberam ser compreensivos nos momentos difíceis e também nos momentos alegres, minha estima e lembrança por estes anos compartilhados.

E a todos, que de algum modo contribuíram para o desenvolvimento e conclusão deste trabalho.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE ABREVIATURAS

OO – ORIENTADA A OBJETOS

POO – PROGRAMAÇÃO ORIENTADA A OBJETOS

UML – LINGUAGEM DE MODELAGEM UNIFICADA

OOSE – *OBJECT-ORIENTED SOFTWARE ENGINEERING*

OMT – *OBJECT MODELING TECHNIQUE*

RESUMO

Este trabalho visa desenvolver uma ferramenta baseada no modelo de diagramas de seqüências da Linguagem Unificada de Modelagem – UML, que são gerados na ferramenta CASE Rational Rose. A partir dos diagramas a ferramenta gera parcialmente código fonte em ambiente de desenvolvimento Delphi. Esta ferramenta tem como propósito simplificar e auxiliar o desenvolvimento de aplicações de programação orientada a objetos.

ABSTRACT

The present work aims to develop a tool based on the model of sequencial diagrams of unified language of modeling – UML, which are generated in CASE tool Rational Rose. From these diagrams on, that the tool to be able to generate source code partially in a DELPHY developed environment. This tool purports to be a way of simplifying and helping the developed of application of oriented objects programming.

1 INTRODUÇÃO

Como toda a técnica de desenvolvimento de sistemas, a orientação a objetos também possui várias abordagens que norteiam o desenvolvimento de projetos de softwares. As abordagens, por exemplo, Rumbaugh, OOSE *(Object-Oriented Software Engineering)*, Booch, OMT (*Object Modeling Technique*), Coad/Yourdon, foram desenvolvidas com o objetivo de melhorar a modelagem e desenvolvimento de sistemas orientados a objeto. De início, recebidas com entusiasmo pelas pessoas envolvidas em desenvolvimento de software orientados a objetos, na medida em que suas grandes diferenças se tornaram aparentes, as questões relativas a qual seria a melhor se tornaram mais freqüentes ([COL1994]).

Conforme [BAR1998], o grande problema do desenvolvimento de novos sistemas utilizando a orientação a objetos nas fases de análise de requisitos e análise de sistemas é que não existe uma notação padronizada e realmente eficaz que abranja qualquer tipo de aplicação que se deseje. Cada simbologia existente possui seus próprios conceitos, gráficos e terminologias, resultando numa grande confusão, especialmente para aqueles que querem utilizar a orientação a objetos não só sabendo para que lado aponta a seta de um relacionamento, mas sabendo criar modelos de qualidade para ajudá-los a construir e manter sistemas cada vez mais eficazes.

As técnicas orientadas a objetos permitem que o software seja construído a partir de objetos que tenham um comportamento específico. Isto pode simplificar o projeto de sistemas complexos e facilitar a manutenção. A tendência indica que o software será montado a partir de componentes e pacotes de vários fabricantes ([SCO1996]).

Segundo [FUR1998], são comuns casos em que se aplica análise baseada em objetos para especificação do sistema, mas a implementação é feita em um ambiente que não suporta a orientação a objetos. Em outros casos, parte-se de uma definição tradicional de sistema e o software é construído com uso de algum mecanismo de orientação a objetos, porém sem respeitar uma arquitetura verdadeiramente orientada a objetos. Em ambas as situações, não se obtêm todos os benefícios associados à orientação a objetos, e as iniciativas acabam sendo consideradas frustrantes ou pouco vantajosas.

A *Unified Modeling Language* (Linguagem de Modelagem Unificada) - UML, merece destaque especial, uma vez que ela foi projetada para servir como uma linguagem de modelagem orientada a objetos, indiferente ao método de desenvolvimento, pois pode substituir – sem perda de informação – as notações dos métodos Booch, OMT e OOSE, entre outros ([MUL1997]).

A UML é muito mais que a padronização de uma notação. É também o desenvolvimento de novos conceitos não normalmente usados. Por isso e muitas outras razões, o bom entendimento da UML não é apenas aprender a simbologia e o seu significado, mas também significa aprender a modelar orientado a objetos no estado da arte. Ela disponibiliza uma série de diagramas para que seja possível levantar a especificação dos requisitos, a estrutura estática, a parte dinâmica ou comportamental, e os aspectos de implementação do sistema ou artefato do sistema ([BAR1997]).

A UML foi desenvolvida por Grady Booch, James Rumbaugh, e Ivar Jacobson. Eles possuem um extenso conhecimento na área de modelagem orientada a objetos, já que as três mais conceituadas metodologias de modelagem orientada a objetos foram eles que desenvolveram, e a UML é a junção do que havia de melhor nestas três metodologias adicionado novos conceitos e visões da linguagem.

Conforme [MUL1997], 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.

1.1 JUSTIFICATIVA

Justifica-se o trabalho pela aplicabilidade da metodologia orientada a objetos para profissionais de desenvolvimento de sistemas, permitindo, assim:

a) fácil compreensão dos benefícios da OO;

b) melhorar o entendimento das metodologias e técnicas existentes;

c) facilitar a geração e posterior reutilização de código fonte a partir de dados modelados para OO.

Como método de especificação, abrangendo principalmente diagramas de seqüência, será usada a UML e para implementação o ambiente de desenvolvimento Delphi.

1.2 OBJETIVO

O objetivo principal do trabalho é desenvolver um protótipo para gerar código fonte a partir de diagramas de seqüências criados na ferramenta Rational Rose, segundo a especificação UML. O ambiente de desenvolvimento do código fonte gerado é o Delphi e a finalidade do protótipo é facilitar a utilização de aplicações orientadas a objetos.

1.3 ESTRUTURA DO TRABALHO

O trabalho está organizado da seguinte forma: no primeiro capítulo, é apresentada uma pequena introdução ao trabalho, a fim de dar ao leitor as informações necessárias ao entendimento do assunto abordado. São apresentados os objetivos e delimitada a abrangência deste, esclarecendo-se sob que ponto de vista será tratado o assunto.

No segundo capítulo, define-se o uso e a importância de uma metodologia no desenvolvimento de sistemas, descreve-se alguns conceitos de Metodologia Orientada a

Objetos, definições da UML, modelos, diagramas e metodologias de desenvolvimento que a compõem.

No terceiro capítulo, uma análise dos diagramas de seqüências, onde tem-se suas principais definições.

No quarto capítulo, breve explanação das principais características existentes nos ambientes de desenvolvimento utilizados para desenvolver o protótipo e a especificação do protótipo.

No quinto capítulo, apresenta-se a especificação e implementação do protótipo.

No sexto e último capítulo, as conclusões sobre o resultado conseguido com o trabalho, dificuldades encontradas e sugestões para aperfeiçoamento do mesmo.

2 UML - LINGUAGEM DE MODELAGEM UNIFICADA

2.1 TECNOLOGIA ORIENTADA A OBJETOS

Segundo [ERI1998], os conceitos da orientação a objetos já vêm sido discutidos há muito tempo, desde o lançamento da 1ª linguagem orientada a objetos, a SIMULA. Vários "papas" da engenharia de software mundial como Peter Coad, Edward Yourdon e Roger Pressman abordaram extensamente a análise orientada a objetos como realmente um grande avanço no desenvolvimento de sistemas.

Conforme [MUL1997], o uso da tecnologia de objetos como metodologia básica para o desenvolvimento de sistemas, abrangendo todo o ciclo desde a análise até a construção de códigos, é uma prática recente. Apenas na década de 80 surgiram os primeiros estudos sobre o uso da OO para especificação de projetos de sistemas, a exemplo dos *papers* de Booch, publicados em 1986.

A partir daí, não foi difícil a utilização dos mesmos conceitos e mecanismos para a análise de sistemas e no final daquela década já surgiam vários ensaios a respeito. Observou-se então uma sucessão de trabalhos sobre técnicas de modelagem OO para especificação de sistemas, podendo ser catalogados dezenas de autores, muitos deles com renome mundial.

Conforme [ERI1998], os conceitos que Coad, Yourdon, Pressman e tantos outros abordaram, discutiram e definiram em suas publicações foram que:

a) a orientação a objetos é uma tecnologia para a produção de modelos que especifiquem o domínio do problema de um sistema;
b) quando construídos corretamente, sistemas orientados a objetos são flexíveis a mudanças, possuem estruturas bem conhecidas e provêm a oportunidade de criar e implementar componentes totalmente reutilizáveis;
c) modelos orientado a objetos são implementados convenientemente utilizando uma linguagem de programação orientada a objetos. A engenharia de software orientada a objetos é muito mais que utilizar mecanismos de sua linguagem de programação,

é saber utilizar da melhor forma possível todas as técnicas da modelagem orientada a objetos;

d) a orientação a objetos não é só teoria, mas uma tecnologia de eficiência e qualidade comprovadas usada em inúmeros projetos e para construção de diferentes tipos de sistemas.

A orientação a objetos requer um método que integre o processo de desenvolvimento e a linguagem de modelagem com a construção de técnicas e ferramentas adequadas.

Segundo [MUL1997], uma metodologia define uma linha para obter-se fidelidade de resultados. Todas as atividades baseadas no conhecimento usam métodos que variam em sofisticação e formalidade. Engenheiros seguem suas diretrizes de engenharia, e assim ocorre com vários profissionais em suas profissões. De semelhante maneira, uma metodologia de desenvolvimento de software descreve como modelar e construir sistemas de software com fidelidade de reprodução.

Em geral, metodologias permitem construir a partir de modelos de elementos, estruturas que constituem os conceitos fundamentais para representação de sistemas. Também definem uma representação, freqüentemente gráfica, que permite a manipulação fácil de modelos e a comunicação e troca de informações entre as várias fases envolvidas. Qualquer metodologia de desenvolvimento de software tem que levar em conta a organização, inter-relações e layouts das estruturas para obter o comportamento macroscópico complexo do sistema que está sendo criado ([MUL1997]).

Um dos fatores críticos encontrados no desenvolvimento de sistemas orientado a objetos, é a falta de definição de uma linguagem metodológica. Um programa orientado a objetos é criado baseado na informação que será processada em vez de basear-se na operação propriamente dita. Isso significa que os dados e as instruções que os manipulam são tratados da mesma forma [(COA1991)].

Conforme [SCO1996] a orientação a objetos permite o reaproveitamento de código. Os objetos armazenados podem ser utilizados em outros aplicativos. Desta maneira, deve-se criar

objetos mais genéricos possíveis, para que a programação possa se assemelhar a uma linha de montagem de componentes pré-fabricados. A técnica promove um aumento de produtividade em programação.

Segundo [ROC1996], as vantagens de se usar objetos como blocos para a construção de aplicações são muitas. Podemos citar:

a) simplicidade: os objetos escondem a complexidade do código. Pode-se criar uma complexa aplicação gráfica usando botões, janelas, barras de rolamento, etc., sem conhecer complexidade do código utilizado para criá-los;
b) reutilização de código: Um objeto, depois de criado, pode ser reutilizado por outras aplicações, ter suas funções extendidas e serem usados como blocos fundamentais em sistemas mais complexos;
c) inclusão dinâmica: objetos podem ser incluídos dinamicamente no programa, durante a execução. Isso permite que vários programas compartilhem os mesmos objetos e classes, reduzindo o seu tamanho final.

2.1.1 CONCEITOS BÁSICOS DA ORIENTAÇÃO A OBJETOS

A tecnologia orientada a objetos fundamenta-se nos seguintes conceitos:

a) objeto: os objetos consistem de modelos (abstrações) de objetos reais. Eles preservam as características essenciais de um objeto real: o seu estado e o seu comportamento ([ROC1996]). Um objeto é qualquer entidade, não necessariamente palpável que apresenta atributos e/ou métodos. Ele deve ser visto como algo que tenha vida, mesmo que seja a abstração de algo inanimado. Esta vida fica caracterizada pelos métodos que o objeto pode executar, este mecanismo é chamado Antropomorfismo ([JUN1998]). De acordo com [COL1994], um objeto corresponde a uma concepção, abstração ou coisa que pode ser identificada distintamente. Durante a análise, objetos tem atributos e podem ser envolvidos em relacionamentos com outros objetos. Durante o projeto, a noção de objeto é estendida pela introdução de métodos e atributos de objetos. Na fase de implementação a noção de objeto é determinada pela linguagem de programação. Pode-se dizer que objeto é uma instância de uma classe;

b) atributos: atributos definem o estado interno de um objeto, suas características, por exemplo, o objeto estante possui as características tamanho, cor, tipo de estrutura. Estes atributos podem ser modificados ao longo da vida do objeto, como esta estante que em determinado momento pode estar vazia, mas se logo após alguns livros forem colocados nela passará a ser uma estante com livros. Aos atributos de um objeto apenas o próprio objeto deve ter acesso ([JUN1998]);

c) métodos: são códigos para implementação em uma classe, ou operação interna, ou seja, o processo de desenvolvimento [COL1994]. São as funções que operam sobre o objeto. Por exemplo, com o objeto caneta podemos escrever e desenhar. São as funções oferecidas pelo objeto caneta. Quando um objeto executa um serviço, dizemos que ele apresenta um determinado comportamento, ou seja, uma reação. O comportamento do objeto será resultado da ação efetuada sobre ele, e da função que esta ação chamou. Por exemplo, tanto o rádio quanto a TV possuem um botão para ligá-los. Quando apertamos este botão (ou seja, efetuamos uma ação sobre este objeto), o comportamento apresentado por ambos é diferente, já que o serviço chamado por esta ação é diferente: ligar o rádio significa emitir som; ligar a TV significa emitir som e imagem. Um evento, ou seja, um estímulo externo, também pode estimular o objeto a executar determinado serviço ([JUN1998]). Objetos de software se comunicam entre si através de mensagens. Como não temos acesso às variáveis a não ser através de métodos, a troca de mensagens consiste da invocação de métodos e o possível retorno de valores ([ROC1996]);

d) classes: consiste de um texto que descreve quais são os atributos e os métodos de todos os objetos pertencentes a esta classe. Chamamos o objeto de instância de uma determinada classe, o que significa que ele possui o comportamento e as características definidos pela classe para suas instâncias, este processo é chamado de instanciação. Todo objeto pertence a uma classe. É conveniente que o nome de uma classe seja colocado no plural, já que uma classe descreve atributos e comportamentos (métodos) de todos os objetos da mesma ([JUN1998]);

e) 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];

f) abstração: abstração é a capacidade de olhar apenas uma parte de um todo, ou seja, retirar da realidade apenas as entidades e fenômenos considerados essenciais, excluindo-se todos os aspectos irrelevantes ou secundários. Através do mecanismo de abstração, o ser humano pode entender formas complexas, ao dividi-las em partes e analisar cada parte separadamente. Por exemplo, para entendermos o funcionamento das plantas, dividimo-las em três partes básicas: raiz, caule e folhas. Estudamos cada uma de suas partes, abstraindo as demais. Para cada parte, identificamos funções diferentes e bem delimitadas. A raiz é responsável pela absorção de água e sais minerais, o caule transporta os materiais coletados para as folhas e estas, através destes materiais fornecidos, produzem, através de fotossíntese, alimento para toda a planta ([JUN1998]);

g) encapsulamento: é o processo de combinar tipos de dados, dados e funções relacionadas em um único bloco de organização e só permitir o acesso a eles através de métodos determinados. Por exemplo, existe um número determinado de coisas que se pode fazer com um toca-fitas. Pode-se avançar, voltar, gravar, tocar, parar, interromper e ejetar a fita. Dentro do toca-fitas, porém, há várias outras funções sendo realizadas como acionar o motor, desligá-lo, acionar o cabeçote de gravação, liberar o cabeçote, e outras operações mais complexas. Essas funções são escondidas dentro do mecanismo do toca-fitas e não temos acesso a elas diretamente. Quando apertamos o *play* o motor é ligado e o cabeçote de reprodução acionado, mas não precisamos saber como isso é feito para usar o toca-fitas. Uma das principais vantagens do encapsulamento é esconder a complexidade do código. Outra, é proteger os dados. Permitindo o acesso a eles apenas através de métodos evita que seus dados sejam corrompidos por aplicações externas. ([ROC1996]);

h) herança: em programação, a herança ocorre quando um objeto aproveita a implementação (estruturas de dados e métodos) de um outro tipo de objeto para desenvolver uma especialização dele. A herança permite a formação de uma hierarquia de classes, onde cada nível é uma especialização do nível anterior, que é mais genérico. É comum desenvolver estruturas genéricas para depois ir acrescentando detalhes em POO (Programação Orientada a Objetos). Isto simplifica o código e permite uma organização maior de um projeto. Também favorece a reutilização de código ([ROC1996]);

i) polimorfismo: é a propriedade de se utilizar um mesmo nome para fazer coisas diferentes. Em programação, uma mesma mensagem poderá provocar um resultado diferente, dependendo dos argumentos que foram passados e do objeto que o receberá.

2.1.2 BENEFÍCIOS DA ORIENTAÇÃO A OBJETOS

Conforme [MAR1994], os sistemas orientados a objetos podem representar melhor o mundo real, uma vez que a percepção e o raciocínio do ser humano estão relacionados diretamente com o conceito de objetos. Este fato permite uma modelagem mais perfeita e natural. Além desta característica, a OO oferece muitos outros benefícios:

a) a mesma notação é usada desde a análise até o projeto e a implementação, de modo que a informação adicionada em uma etapa do desenvolvimento não é necessariamente perdida ou traduzida para a etapa seguinte ([RUM1994]);
b) esta abordagem incentiva os desenvolvedores a trabalharem e pensarem em termos do domínio da aplicação durante a maior parte do ciclo de vida, ou seja, dedicação maior à fase de análise. A parte mais difícil do desenvolvimento de software é a manipulação de sua essência face à inerente complexidade do problema, e não os acidentes de seu mapeamento em uma determinada linguagem, que são devidos a imperfeições temporárias das ferramentas, que estão sendo rapidamente corrigidas;
c) ocorre uma redução na quantidade de erros com conseqüente diminuição do tempo despendido nas etapas de codificação e teste, visto que os problemas são detectados mais cedo e corrigidos antes da implementação;
d) melhora a comunicação entre desenvolvedores e usuários já que o sistema refletirá o modelo do negócio. Os modelos na orientação a objetos espelham a estrutura e o comportamento dos objetos do negócio, diminuindo o abismo existente nas outras abordagens que tratam dados e funções separadas;
e) no desenvolvimento baseado em objetos há uma redução no tempo de manutenção, pois as revisões são mais fáceis e mais rápidas já que o problema é mais bem localizado;

f) favorece a reutilização, já que ocorre a construção de componentes mais gerais, estáveis e independentes;
g) a cada dia os sistemas estão mais complexos e sujeitos a alterações freqüentes; esta tecnologia suporta muito bem a construção destes tipos de sistemas;
h) facilidade de extensão, visto que objetos têm sua interface bem definida. A criação de novos objetos que se comunicam com os já existentes não obriga o desenvolvedor a conhecer o interior destes últimos.

Como o desenvolvimento de sistemas está intimamente relacionado com o negócio das empresas, acaba por herdar as mesmas exigências. Flexibilidade, produtividade e qualidade são exatamente o que a orientação a objetos vem a oferecer. Produtividade pela facilidade de reutilização a partir de componentes já desenvolvidos ou adquiridos no mercado para encaixá-los na sua aplicação. Qualidade pelo fato do encapsulamento reduzir a interferência múltipla entre os módulos do sistema. Como as dependências entre os objetos tendem a não existir, reduz-se a probabilidade de uma alteração em uma classe de objetos introduzir erro em outras. E como os componentes são utilizados por várias aplicações, demonstram boa qualidade pois já foram testados exaustivamente. Já a flexibilidade é obtida através dos mecanismos de herança e polimorfismo, onde desenvolvem-se sistemas com menor redundância de código, acelerando o processo através do aproveitamento de código existente e facilitando a manutenção.

2.2 UML - A UNIFICAÇÃO DOS MÉTODOS PARA A CRIAÇÃO DE UM NOVO PADRÃO

Conforme [BAR1998], a UML é uma tentativa de padronizar a modelagem orientada a objetos de uma forma que qualquer sistema, seja qual for o tipo, possa ser modelado corretamente, com consistência, fácil de se comunicar com outras aplicações, simples de ser atualizado e compreensível.

Existem várias metodologias de modelagem orientada a objetos que até o surgimento da UML causavam uma “guerra” entre a comunidade de desenvolvedores orientado a objetos. A UML acabou com esta “guerra” trazendo as melhores idéias de cada uma destas metodologias, e mostrando como deveria ser a migração de cada uma para a UML.

Conforme [RAT1997], a UML é uma linguagem padrão para especificar, visualizar, documentar e construir artefatos de um sistema, cujos objetivos são:

a) a modelagem de sistemas usando conceitos da orientação a objetos;
b) estabelecer uma união a fim de que métodos conceituais sejam também executáveis;
c) criar uma linguagem de modelagem usável tanto pelo homem como pela máquina.

2.3 USO DA UML

Segundo [FUR1998], a UML pode ser usada para:

a) mostrar as fronteiras de um sistema e suas funções principais utilizando atores e casos de uso;
b) ilustrar a realização de casos de uso com diagramas de interação;
c) representar uma estrutura estática de um sistema utilizando diagramas de classes;
d) modelar o comportamento de objetos através de diagramas de transição de estado;
e) revelar a arquitetura de implementação física com diagramas de componentes e de implantação;
f) estender sua funcionalidade através de estereótipos.

Conforme [BAR1998], a UML é usada no desenvolvimento dos mais diversos tipos de sistemas. Ela abrange sempre qualquer característica de um sistema em um de seus diagramas e é também aplicada em diferentes fases do desenvolvimento de um sistema, desde a especificação da análise de requisitos até a finalização com a fase de testes.

O objetivo da UML é descrever qualquer tipo de sistema, em termos de diagramas orientado a objetos. Naturalmente, o uso mais comum é para criar modelos de sistemas de software, mas a UML também é usada para representar sistemas mecânicos sem nenhum software. Aqui estão alguns tipos diferentes de sistemas com suas características mais comuns:

a) sistemas de informação – armazenar, pesquisar, editar e mostrar informações para os usuários. Manter grandes quantidades de dados com relacionamentos complexos, que são guardados em bancos de dados relacionais ou orientados a objetos;

b) sistemas técnicos – manter e controlar equipamentos técnicos como de telecomunicações, equipamentos militares ou processos industriais. Eles devem possuir interfaces especiais do equipamento e menos programação de software de que os sistemas de informação. Sistemas Técnicos são geralmente sistemas real-time;
c) sistemas real-time integrados – executados em simples peças de hardware integrados a telefones celulares, carros, alarmes etc. Estes sistemas implementam programação de baixo nível e requerem suporte real-time;
d) sistemas distribuídos – distribuídos em máquinas onde os dados são transferidos facilmente de uma máquina para outra. Eles requerem mecanismos de comunicação sincronizados para garantir a integridade dos dados e geralmente são construídos em mecanismos de objetos como CORBA, COM/DCOM ou Java Beans/RMI;
e) sistemas de software – definem uma infra-estrutura técnica que outros softwares utilizam. Sistemas Operacionais, bancos de dados, e ações de usuários que executam ações de baixo nível no hardware, ao mesmo tempo que disponibilizam interfaces genéricas de uso de outros softwares;
f) sistemas de negócios – descreve os objetivos, especificações (pessoas, computadores etc.), as regras (leis, estratégias de negócios etc.), e o atual trabalho desempenhado nos processos do negócio ([BAR1998]).

2.4 FASES DO DESENVOLVIMENTO DE UM SISTEMA EM UML

Conforme [RAT1997], são cinco as fases no desenvolvimento de sistemas de software:

a) análise de requisitos;
b) análise;
c) design;
d) programação;
e) testes.

As fases do desenvolvimento de sistemas de software em UML são descritas a seguir.

2.4.1 ANÁLISE DE REQUISITOS

Esta fase captura as intenções e necessidades dos usuários do sistema a ser desenvolvido através do uso de funções chamadas *use-cases* ou casos de uso. Através do desenvolvimento de casos de uso, as entidades externas ao sistema (em UML chamados de "atores externos") que interagem e possuem interesse no sistema são modelados entre as funções que eles requerem, funções estas chamadas de casos de uso. Os atores externos e os casos de uso são modelados com relacionamentos que possuem comunicação associativa entre eles ou são desmembrados em hierarquia. Cada caso de uso modelado é descrito através de um texto, e este especifica os requerimentos do ator externo que utilizará este caso de uso. O diagrama de casos de uso mostrará o que os atores externos, ou seja, os usuários do futuro sistema deverão esperar do aplicativo, conhecendo toda sua funcionalidade sem importar como esta será implementada. A análise de requisitos também pode ser desenvolvida baseada em processos de negócios, e não apenas para sistemas de software.

2.4.2 ANÁLISE

A fase de análise está preocupada com as primeiras abstrações (classes e objetos) e mecanismos que estarão presentes no domínio do problema. As classes são modeladas e ligadas através de relacionamentos com outras classes, e são descritas no Diagrama de Classe. As colaborações entre classes também são mostradas neste diagrama para desenvolver os casos de uso modelados anteriormente, estas colaborações são criadas através de modelos dinâmicos em UML. Na análise, só serão modeladas classes que pertençam ao domínio principal do problema do software, ou seja, classes técnicas que gerenciem banco de dados, interface, comunicação, concorrência e outros não estarão presentes neste diagrama.

2.4.3 DESIGN

Na fase de design (projeto), o resultado da análise é expandido em soluções técnicas. Novas classes serão adicionadas para prover uma infra-estrutura técnica: a interface do usuário e de periféricos, gerenciamento de banco de dados, comunicação com outros sistemas, dentre outros. As classes do domínio do problema modeladas na fase de análise são mescladas nessa

nova infra-estrutura técnica tornando possível alterar tanto o domínio do problema quanto a infra-estrutura. O design resulta no detalhamento das especificações para a fase de programação do sistema.

2.4.4 PROGRAMAÇÃO

Na fase de programação, as classes provenientes do design são convertidas para o código da linguagem orientada a objetos escolhida (a utilização de linguagens procedurais é extremamente não recomendada). Dependendo da capacidade da linguagem usada, essa conversão pode ser uma tarefa fácil ou muito complicada. No momento da criação de modelos de análise e design em UML, é melhor evitar traduzi-los mentalmente em código. Nas fases anteriores, os modelos criados são o significado do entendimento e da estrutura do sistema, então, no momento da geração do código onde o analista conclua antecipadamente sobre modificações em seu conteúdo, seus modelos não estarão mais demonstrando o real perfil do sistema. A programação é uma fase separada e distinta onde os modelos criados são convertidos em código.

2.4.5 TESTES

Um sistema normalmente é executado em testes de unidade, integração, e aceitação. Os testes de unidade são para classes individuais ou grupos de classes e são geralmente testados pelo programador. Os testes de integração são aplicados já usando as classes e componentes integrados para se confirmar se as classes estão cooperando uma com as outras como especificado nos modelos. Os testes de aceitação observam o sistema e verificam se o sistema está funcionando como o especificado nos primeiros diagramas de casos de uso.

O sistema será testado pelo usuário final e verificará se os resultados mostrados estão realmente de acordo com as intenções do usuário final.

Estas cinco fases não devem ser executadas necessariamente na ordem descrita, mas concomitantemente de forma que problemas detectados numa certa fase modifiquem e

melhorem as fases desenvolvidas anteriormente de forma que o resultado global gere um produto de alta qualidade e performance.

Segundo [FUR1998], a UML pode ser dividida esquematicamente em duas categorias de diagramas: estáticos e dinâmicos (funcionais). O estudo concentrou-se nos diagramas dinâmicos e a implementação especificamente no diagrama de seqüência.

2.5 A NOTAÇÃO DA UML

Tendo em mente as cinco fases do desenvolvimento de softwares, as fases de análise de requisitos, análise e projeto utilizam em seu desenvolvimento cinco tipos de visões, nove tipos de diagramas e vários modelos de elementos que serão utilizados na criação dos diagramas e mecanismos gerais, que todos em conjunto especificam e exemplificam a definição do sistema, tanto como a definição no que diz respeito à funcionalidade estática e dinâmica do desenvolvimento de um sistema ([BAR1998]).

Segundo [BAR1998], as partes que compõem a UML são:

a) visões: as visões mostram diferentes aspectos do sistema que está sendo modelado. A visão não é um gráfico, mas uma abstração consistindo em uma série de diagramas. Definindo um número de visões, cada uma mostrará aspectos particulares do sistema, dando enfoque a ângulos e níveis de abstrações diferentes e uma figura completa do sistema poderá ser construída. As visões também podem servir de ligação entre a linguagem de modelagem e o método/processo de desenvolvimento escolhido;
b) modelos de elementos: os conceitos usados nos diagramas são modelos de elementos que representam definições comuns da orientação a objetos como as classes, objetos, mensagem, relacionamentos entre classes incluindo associações, dependências e heranças;
c) mecanismos gerais: os mecanismos gerais provém comentários suplementares, informações, ou semântica sobre os elementos que compõem os modelos; eles provém também mecanismos de extensão para adaptar ou estender a UML para um método/processo, organização ou usuário específico;

d) diagramas: os diagramas são os gráficos que descrevem o conteúdo em uma visão. UML possui nove tipos de diagramas que são usados em combinação para prover todas as visões do sistema.

Serão discorridos, agora, alguns aspectos de cada parte componente da UML.

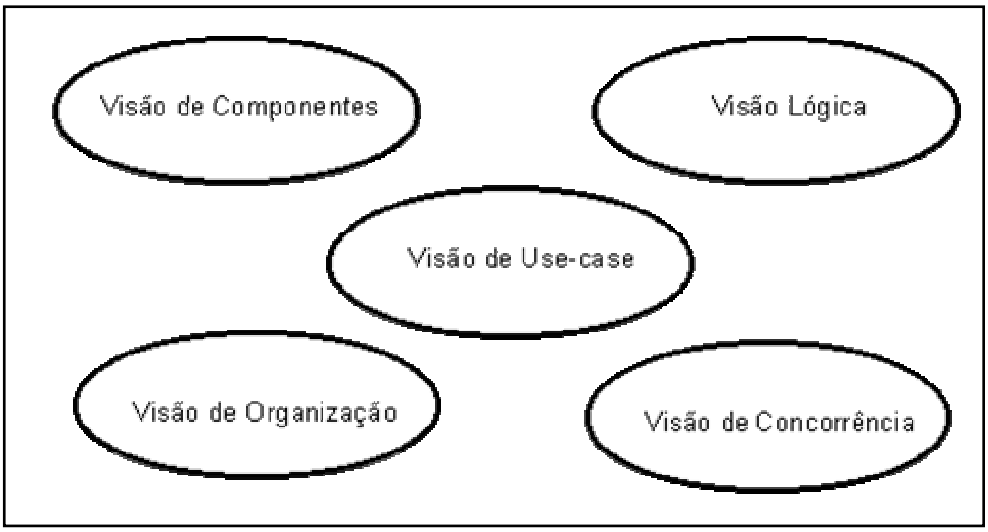
2.5.1 VISÕES

O desenvolvimento de um sistema complexo não é uma tarefa fácil. O ideal seria que o sistema inteiro pudesse ser descrito em um único gráfico e que este representasse por completo as reais intenções do sistema sem ambigüidades, sendo facilmente interpretável. Infelizmente, isso é impossível. Um único gráfico é incapaz de capturar todas as informações necessárias para descrever um sistema.

Um sistema é composto por diversos aspectos: funcional (que é sua estrutura estática e suas interações dinâmicas), não funcional (requisitos de tempo, confiabilidade, desenvolvimento) e aspectos organizacionais (organização do trabalho, mapeamento dos módulos de código). Então o sistema é descrito em um certo número de visões, cada uma representando uma projeção da descrição completa e mostrando aspectos particulares do sistema.

Cada visão é descrita por um número de diagramas que contém informações que dão ênfase aos aspectos particulares do sistema. Existe em alguns casos uma certa sobreposição entre os diagramas o que significa que um deste pode fazer parte de mais de uma visão. Os diagramas que compõem as visões contém os modelos de elementos do sistema. As visões que compõem um sistema são (figura1):

Figura 1: representação gráfica das visões que compõe a UML



Fonte: [BAR1998]

a) visão *use-case*: descreve a funcionalidade do sistema desempenhada pelos atores externos do sistema (usuários). A visão *use-case* é central, já que seu conteúdo é base do desenvolvimento das outras visões do sistema. Essa visão é montada sobre os diagramas de *use-case* e eventualmente diagramas de atividade;
b) visão lógica: descreve como a funcionalidade do sistema será implementada. É feita principalmente pelos analistas e desenvolvedores. Em contraste com a visão *use-case*, a visão lógica observa e estuda o sistema internamente. Ela descreve e especifica a estrutura estática do sistema (classes, objetos e relacionamentos) e as colaborações dinâmicas quando os objetos enviarem mensagens uns para os outros para realizarem as funções do sistema. Propriedades como persistência e concorrência são definidas nesta fase, bem como as interfaces e as estruturas de classes. A estrutura estática é descrita pelos diagramas de classes e objetos. O modelamento dinâmico é descrito pelos diagramas de estado, seqüência, colaboração e atividade;
c) visão de componentes: é uma descrição da implementação dos módulos e suas dependências. É principalmente executado por desenvolvedores, e consiste nos componentes dos diagramas;
d) visão de concorrência: trata a divisão do sistema em processos e processadores. Este aspecto, que é uma propriedade não funcional do sistema, permite uma melhor

utilização do ambiente onde o sistema se encontrará, se o mesmo possui execuções paralelas, e se existe dentro do sistema um gerenciamento de eventos assíncronos. Uma vez dividido o sistema em linhas de execução de processos concorrentes (*threads*), esta visão de concorrência deverá mostrar como se dá a comunicação e a concorrência destas *threads*. A visão de concorrência é suportada pelos diagramas dinâmicos, que são os diagramas de estado, seqüência, colaboração e atividade, e pelos diagramas de implementação, que são os diagramas de componente e execução;

e) visão de organização: finalmente, a visão de organização mostra a organização física do sistema, os computadores, os periféricos e como eles se conectam entre si. Esta visão será executada pelos desenvolvedores, integradores e testadores, e será representada pelo diagrama de execução.

2.5.2 MODELOS DE ELEMENTOS

A UML está destinada a ser dominante, a linguagem de modelagem comum a ser usada nas indústrias. Ela está totalmente baseada em conceitos e padrões extensivamente testados provenientes das metodologias existentes anteriormente, e também é muito bem documentada com toda a especificação da semântica da linguagem representada em meta-modelos.

Segundo [FOW1997], o meta-modelo da UML é usado como um guia de referência para construir ferramentas, e para compartilhar modelos entre ferramentas diferentes. Um modelo é uma descrição abstrata de um sistema ou um processo (uma representação simplificada que promove compreendimento e capacidade de simulação). A forma do modelo depende do meta-modelo. Funcionalmente decompõe-se o modelo em funções que são mais simples de implementar. Cada meta-modelo define elementos modelo e regras para a composição destes.

O conteúdo do modelo depende do problema. A modelagem com a UML é suficientemente genérica para ser usada em todo o domínio da engenharia de software, além de poder ser aplicado a análise de negócios ([MUL1997]).

Para melhor entendimento da metodologia de desenvolvimento da UML, será abordado de forma geral características de alguns modelos de elementos ([BAR1998]).

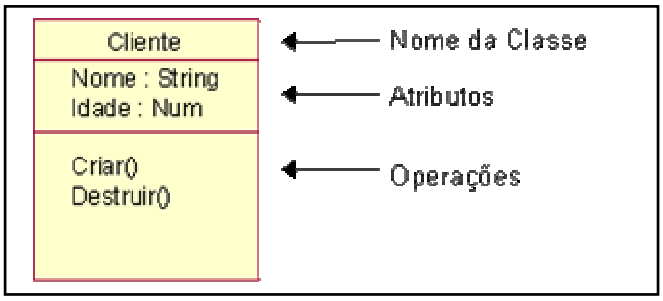
2.5.2.1 CLASSES

Uma classe é a descrição de um tipo de objeto. Todos os objetos são instâncias de classes, onde a classe descreve as propriedades e comportamentos daquele objeto. Objetos só podem ser instanciados de classes. Usam-se classes para classificar os objetos que identificamos no mundo real.

Uma classe pode ser a descrição de um objeto em qualquer tipo de sistema – sistemas de informação, técnicos, integrados real-time, distribuídos ou de software. Num sistema de software, por exemplo, existem classes que representam entidades de software num sistema operacional como arquivos, programas executáveis, janelas, barras de rolagem, etc.

Em UML as classes são representadas por um retângulo dividido em três compartimentos, como exemplifica a figura 2: o compartimento de nome, que conterá apenas o nome da classe modelada, o de atributos, que possuirá a relação de atributos que a classe possui em sua estrutura interna, e o compartimento de operações, que serão o métodos de manipulação de dados e de comunicação de uma classe com outras do sistema. A sintaxe usada em cada um destes compartimentos é independente de qualquer linguagem de programação, embora possa ser usadas outras sintaxes como a do C++, Java, e outras.

Figura 2: representação de uma classe



Fonte: [BAR1998]

2.5.2.2 OBJETOS

Um objeto é um elemento que podemos manipular, acompanhar seu comportamento, criar, destruir. Um objeto existe no mundo real. Pode ser uma parte de qualquer tipo de sistema, por exemplo, uma máquina, uma organização, ou negócio. Existem objetos que não encontramos no mundo real, mas que podem ser vistos de derivações de estudos da estrutura e comportamento de outros objetos do mundo real.

Em UML um objeto é mostrado como uma classe só que seu nome (do objeto) é sublinhado, e o nome do objeto pode ser mostrado opcionalmente precedido do nome da classe.

2.5.2.3 ESTADOS

Todos os objetos possuem um estado que significa o resultado de atividades executadas pelo objeto, e é normalmente determinada pelos valores de seus atributos e ligações com outros objetos.

Um objeto muda de estado quando acontece algo, o fato de acontecer alguma coisa com o objeto é chamado de evento. Através da análise da mudança de estados dos tipos de objetos de um sistema, podemos prever todos os possíveis comportamentos de um objeto de acordo com os eventos que o mesmo possa sofrer.

2.5.2.4 PACOTES

Pacote é um mecanismo de agrupamento, onde todos os modelos de elementos podem ser agrupados. Em UML, um pacote é definido como um mecanismo de propósito geral para organizar elementos semanticamente relacionados em grupos ([RAT1997]). Todos os modelos de elementos que são ligados ou referenciados por um pacote são chamados de conteúdo do pacote. Um pacote possui vários modelos de elementos, e isto significa que estes não podem ser incluídos em outros pacotes.

2.5.2.5 COMPONENTES

Um componente pode ser tanto um código em linguagem de programação como um código executável já compilado. Por exemplo, em um sistema desenvolvido cada arquivo é um componente do sistema, e será mostrado no diagrama de componentes que os utiliza.

2.5.3 RELACIONAMENTOS

Os relacionamentos ligam as classes/objetos entre si criando relações lógicas entre estas entidades. Os relacionamentos podem ser dos seguintes tipos:

a) associação: é uma conexão entre classes, e também significa que é uma conexão entre objetos daquelas classes. Em UML, uma associação é definida como um relacionamento que descreve uma série de ligações, onde a ligação é definida como a semântica entre as duplas de objetos ligados;
b) generalização: é um relacionamento de um elemento mais geral e outro mais específico. O elemento mais específico pode conter apenas informações adicionais. Uma instância (um objeto é uma instância de uma classe) do elemento mais específico pode ser usada onde o elemento mais geral seja permitido;
c) dependência e refinamentos: dependência é um relacionamento entre elementos, um independente e outro dependente. Uma modificação é um elemento independente que afetará diretamente elementos dependentes do anterior. Refinamento é um relacionamento entre duas descrições de uma mesma entidade, mas em níveis diferentes de abstração.

Conforme [RAT1997], um modelo relaciona-se à fase específica do desenvolvimento, e é construído de elementos modelo com as representações associadas diferentes entre eles. Os modelos são manipulados por meio de representações gráficas que são projeções dos elementos contidos em um ou mais modelos. Podem ser construídas de diferentes perspectivas para um modelo básico – cada uma pode mostrar tudo ou parte do modelo, e cada uma possui um diagrama correspondente. A UML define nove diferentes tipos de diagramas, sendo que o escopo do trabalho é direcionado ao diagrama de seqüência, representação temporal de objetos e a interação entre eles.

2.5.4 DIAGRAMAS

Conforme [BAR1998], 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 modelamentos dinâmicos são suportados pelos diagramas de estado, seqüências, colaboração e atividade. E o modelamento funcional é suportado pelos diagramas de componentes e execução.

a) diagrama de casos de uso: 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;

b) 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 superficie de duas dimensões mostrando uma coleção de elementos declarativos de modelo, como classes, tipos e seus respectivos conteúdos e relações;

c) 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;

d) diagrama de estados: 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;

e) diagrama de seqüências: um diagrama de sequências 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 sequência de mensagens enviadas entre os objetos. Como parte principal do desenvolvimento deste protótipo, o diagrama de seqüências será abordado mais adiante;

f) 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 sequência para evidenciar a sequência de mensagens;

g) 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;

h) diagrama de componentes: segundo [ERI1998], o diagrama de componentes 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;

i) 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.

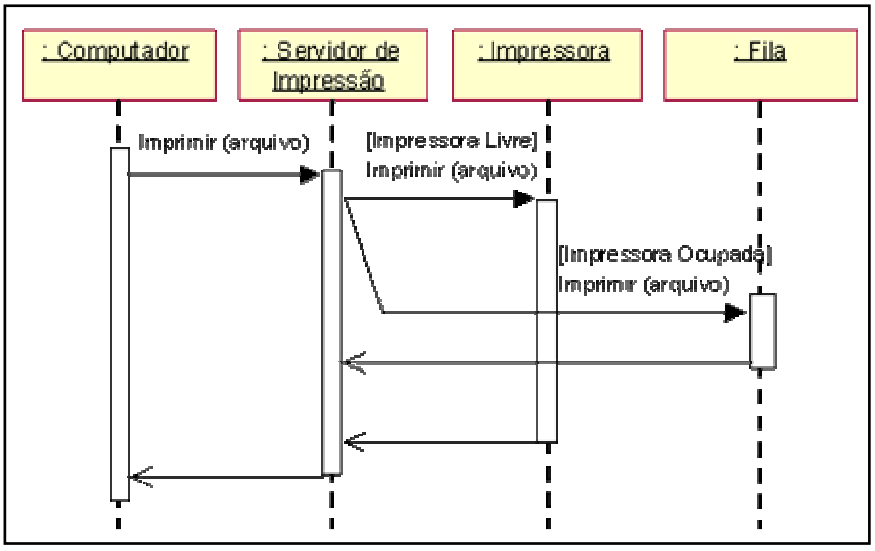
3 DIAGRAMAS DE SEQÜÊNCIAS

Conforme [BAR1998], 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.

Uma interação é implementada por um grupo de objetos que colaboram trocando mensagens. Estas mensagens são representadas ao longo dos vínculos que ligam os objetos e usam setas direcionadas ao receptor da mensagem.

Um diagrama de seqüência representa uma interação entre objetos, enfocado na transmissão de mensagens seqüenciais. Um objeto é representado por um retângulo e uma barra vertical chamada de linha da vida do objeto, conforme figura 3.

Figura 3: exemplo de impressão de arquivo mostrando componentes
do diagrama de seqüência



fonte: [RAT1997]

Se for usado o exemplo da figura 3 no protótipo, poderia gerar um fonte com alguns “buracos”, podendo ocorrer até que o fonte gerado apresente erros de compilação. Se faz importante salientar que o protótipo não é um compilador, e caso o diagrama de seqüências estiver modelado de forma inadequada o arquivo fonte gerado pelo gerador de código do protótipo pode ter erros de compilação. No caso do exemplo os problemas podem ocorrer porque nem todas as mensagens de ativação dos objetos estão devidamente identificadas.

Conforme [FUR1998], diagramas de seqüências 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 cita-se: mensagens recebidas ou enviadas e ativação de objetos. A comunicação entre os objetos é representada como linha com setas horizontais simbolizando as mensagens entre as linhas de vida dos objetos. A seta especifica se a mensagem é síncrona, assíncrona ou simples. As mensagens podem possuir também números seqüenciais, eles são utilizados para tornar mais explícito as seqüências no diagrama.

Em alguns sistemas, objetos são executados concorrentemente, cada um com sua linha de execução (*thread*). Se o sistema usa linhas concorrentes de controle, isto é mostrado como ativação, mensagens assíncronas, ou objetos assíncronos.

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.

Na modelagem orientada a objetos, diagramas de seqüência são usados de dois modos muito diferentes, de acordo com a fase e o nível de detalhe desejado:

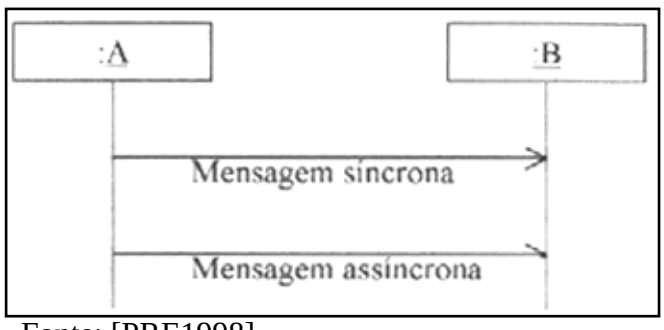
a) o primeiro uso corresponde à documentação de casos de uso. Enfoca a descrição da interação, freqüentemente em situações ocultas ao usuário, sem entrar nos detalhes de sincronização. Neste caso, a informação levada pelas setas corresponde a eventos que acontecem dentro do domínio da aplicação. Nesta fase da modelagem, as setas não correspondem ainda à transmissão de mensagens no sentido de uma linguagem de programação, e a diferença entre controle de fluxos e fluxo de dados geralmente não é estabelecida;
b) o segundo, permite a representação precisa de interações entre objetos. O conceito de uma mensagem une todos os tipos de comunicação entre objetos tais como: chamadas de procedimentos, eventos descontínuos, sinais entre fluxos de execução e interrupções de hardware.

Os diagramas de seqüência distinguem duas categorias principais de transmissão de mensagem:

a) transmissão síncrona: o transmissor é bloqueado e espera até que o objeto chamado termine de processar a mensagem;
b) transmissão assíncrona: o transmissor é bloqueado e pode continuar executando.

Uma transmissão síncrona é representada por uma seta do transmissor da mensagem até seu receptor. Uma transmissão assíncrona é representada por uma meia seta, como identificadas na figura 4.

Figura 4: exemplo de tipos de transmissão de mensagens



Fonte: [PRE1998]

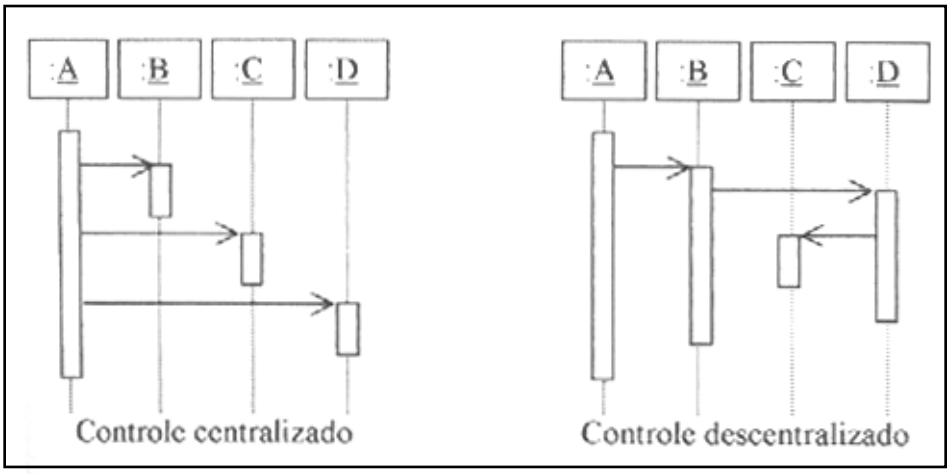
A seta que simboliza uma mensagem, pode ser puxada diagonalmente, para representar demoras de transmissão que não são desprezíveis à dinâmica global de aplicação. Um objeto também pode enviar uma mensagem para si. Esta situação é representada por uma seta de retorno ao longo da linha da vida do objeto.

Os diagramas de seqüência também permitem a representação de ativações de objetos. Uma ativação corresponde ao tempo durante o qual um objeto executa uma ação, ou diretamente ou por outro objeto que o usa como um intermediário. As ativações são representadas através de faixas retangulares, posicionadas ao longo da linha da vida. O começo e o fim de uma faixa correspondem respectivamente ao começo e o fim de uma ativação.

3.1 ESTRUTURAS DE CONTROLE

As formas do diagrama de seqüência refletem as escolhas da estrutura de controle. O diagrama pode ser representado de modo centralizado ou descentralizado. A figura 5 exemplifica os tipos de controle.

Figura 5: formas de controle



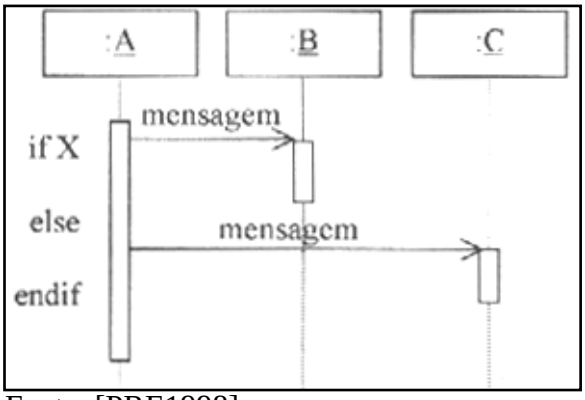
Fonte: [PRE1998]

Os diagramas de seqüência podem ser complementados através de notas textuais, expressas como uma descrição qualquer ou um pseudocódigo. O local de emissão de uma mensagem, chamado de transição, pode ser usado com ponto para identificação do texto. Este nome pode ser usado como uma referência no diagrama.

A adição de pseudocódigo no lado esquerdo do diagrama habilita a representação de laços. O laço de repetição pode também ser representado usando-se uma condição de repetição, posicionado diretamente na mensagem. A repetição é simbolizada pelo caracter *, colocado na frente da condição, entre colchetes.

Assim como laços, condições também podem ser representadas usando pseudocódigos, escritos do lado esquerdo do diagrama. A figura 6 exemplifica o uso de um pseudocódigo.

Figura 6: uso de pseudocódigo. A condição
depende do valor de X



Fonte: [PRE1998]

Como referenciado anteriormente, as condições colocadas na frente das mensagens podem ser substituídas por pseudocódigos. Todas as partes podem também ser representadas por várias setas que originam no mesmo instante, e podem ser distinguidas por condições colocadas na frente das mensagens. Para cada parte, as condições devem ser mutuamente exclusivas. Os pseudocódigos podem ser adaptados a realidade do usuário ou do analista, tornando o uso da notação mais genérica.

4 FERRAMENTAS UTILIZADAS

4.1 RATIONAL ROSE

Segundo [RAT1997], Rational Rose é uma ferramenta de modelagem visual para projetar e criar aplicações de software utilizando padrão UML. Aplicações de software contemporâneas estão crescendo em complexidade. Isto significa que aquele software criado já não pode ser considerado um eufemismo, precisa se tornar uma real disciplina de engenharia.

Rational Rose 98 Edição de Empreendimento é uma das primeiras ferramentas para modelar aplicações de software. A versão 98 é uma versão aperfeiçoada significativa do Rose versão 4.0 e surge principalmente da adoção pelo Grupo de Administração de Objeto de UML. Com ferramentas de apoio embutidas para Visual Basic, Java, Oracle8, C++, Delphi e outras, a ferramenta Rational Rose pode ser usada por quase qualquer usuário.

A Edição de Empreendimento é um produto grande, complexo, caro que requer entendimento significativo de software, criando e modelando conceitos. Suas características flexíveis fazem construir um modelo que gere as classes e o código de apoio para implementar o modelo no idioma de sua escolha, modificando o código em uma ferramenta de desenvolvimento como o Delphi.

4.2 AMBIENTE DE DESENVOLVIMENTO DELPHI

Os aplicativos desenvolvidos para os ambientes corporativos possuem necessidades específicas, quase sempre desenvolvidos com cronogramas apertados e por grupos pequenos de programadores. Deve-se levar em consideração a questão do reaproveitamento de código. A possibilidade de recompilação do código-fonte com modificações mínimas pode facilmente se tornar um aspecto definitivo.

O desenvolvimento do protótipo foi baseado no ambiente de desenvolvimento Borland Delphi 3.0. A escolha deveu-se basicamente devido às facilidades que este ambiente dispõe para geração de componentes.

Dentre os principais recursos do ambiente de desenvolvimento Borland Delphi 3.0 pode-se destacar:

a) possibilidade de geração de executáveis em código nativo;
b) facilita depuração de controles e bibliotecas;
c) integração com a ferramenta case Rational Rose 98, tornando possível a geração de código fonte em Delphi.

Antes do Delphi e do Visual Basic, todas as linguagens de programação eram parecidas do ponto de vista conceitual. Havia diferenças de sintaxe, é claro, bem como diferenças importantes de paradigmas, mas a metodologia da programação em C, por exemplo, era a mesma da programação em Pascal, Cobol ou Fortran.

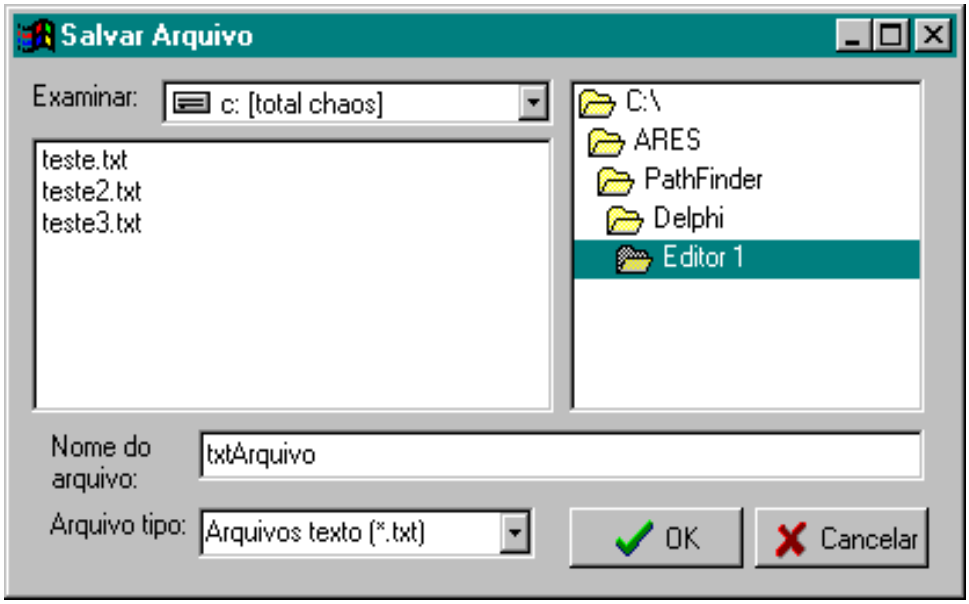
Conforme [CAN1998], as linguagens visuais introduziram estratégias radicalmente novas de programação. O fato é que, com o passar do tempo, escrever programas passou a ser cada vez mais difícil, especialmente programas que exigiam interface gráfica. Entretanto, alguns programadores perceberam que muitas coisas que eram difíceis de ser feitas, como construir janelas, menus ou botões, podiam ser feitas sempre da mesma forma. Estes programadores, que já tinham o hábito de colecionar sub-rotinas de utilização geral, passaram a encapsular algumas destas rotinas em uma espécie de "objeto" pronto para ser usado. A idéia final, que deu origem ao Visual Basic e ao Delphi, foi a percepção de que vários destes objetos

podiam simplesmente ser desenhados na tela como se desenha um retângulo ou outra figura qualquer.

O Visual Basic surgiu no começo da década de 90 e deu início a uma profusão de linguagens visuais, tais como Delphi, Visual C, Visual Fox Pro, e outras. É difícil escolher entre Visual Basic e Delphi, por exemplo (uma 'briga' que parece estar muito em moda). Tudo dependerá de que tipo de aplicativo você estiver desenvolvendo, mas de certa forma é sempre bom saber um pouco de cada linguagem.

O Delphi é um pacote de ferramentas de programação concebido para programação em Windows. Os objetos são desenhados na tela de forma visual, com auxílio do mouse, e não por meio de programação. A programação em si é orientada a eventos. Quando um evento ocorre, tal como uma tecla pressionada ou um clique de mouse, uma mensagem é enviada para a fila de mensagens do Windows. A mensagem estará disponível para todos os aplicativos que estiverem rodando, mas apenas aquele interessado no evento responderá à mensagem. Tudo que o usuário precisa fazer é detectar o evento e mandar que um trecho de código seja executado quando isto acontecer. O Delphi torna esta tarefa relativamente fácil.

Figura 7: Ambiente de Desenvolvimento Delphi



5 ESPECIFICAÇÃO DO PROTÓTIPO

Para realização do protótipo, fez-se necessário cumprir as seguintes etapas:

a) modelagem: criou-se um diagrama de classes de um sistema hipotético e o respectivo diagrama de seqüências das ações a serem executadas pelo sistema. Esta etapa foi desenvolvida na ferramenta de modelagem visual Rational Rose;
b) gravação da modelagem: em um arquivo extensão .mdl da própria ferramenta Rational Rose foram gravados os modelos de diagramas criados. Este arquivo pode ser lido por qualquer editor;
c) depuração do arquivo de modelagem: o arquivo gerado é depurado linha a linha. O protótipo procura por algumas “palavras chaves” que identificam nome do arquivo, início e final de um bloco referente a uma classe, assim como a mensagem (função). As informações sobre as classes e operações são retiradas do arquivo utilizado como origem, que é o arquivo de extensão .mdl;
d) gerar código fonte: após toda a depuração do arquivo de modelagem concluir-se e todos os dados de relevante importância para o protótipo serem extraídos, o código fonte é gerado e gravado em um arquivo extensão .pas. no ambiente de desenvolvimento Delphi, ficando pronto para posteriores aplicações;
e) visualizar e editar código fonte: é disponibilizado ao desenvolvedor a visualização do código fonte para verificação e também sua edição, quando para ele for necessária.

Lembra-se que a etapa de depuração do arquivo de modelagem e as próximas etapas descritas são totalmente desenvolvidas no ambiente de desenvolvimento Delphi.

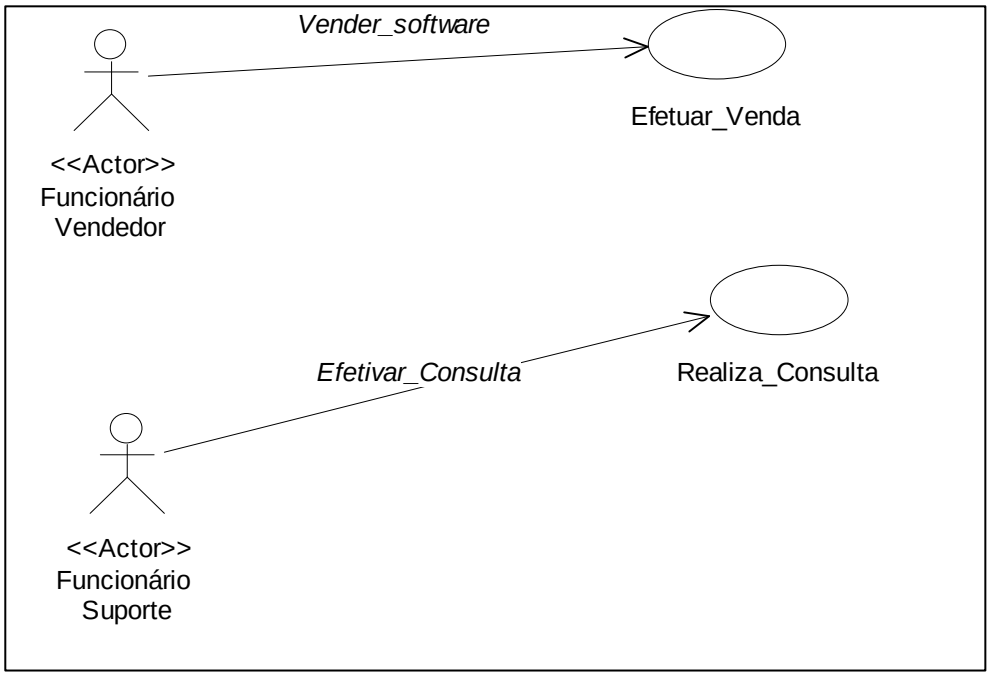
5.1 IMPLEMENTAÇÃO DO PROTÓTIPO

O escopo deste protótipo é a geração de código-fonte a partir de diagramas de seqüências criados para um sistema. Para exemplificar, será efetuado um estudo de caso de um sistema hipotético simples de uma empresa que vende pacotes de softwares. A figura 8 mostra a fase inicial do protótipo, com a construção do diagrama de casos de uso. Um conjunto de casos de uso é importante para se compreender o que o usuário quer. Um caso de uso descreve uma funcionalidade a ser oferecida pelo sistema, ou seja, um serviço prestado ao usuário.

A denominação utilizada para os casos de uso é iniciada com um verbo no infinitivo que expressa a tarefa do usuário que o sistema irá apoiar. O primeiro caso de uso da figura 8 mostra o exemplo Vender Software - este caso de uso apoiará a tomada de decisão do funcionário vendedor para que ele cumpra sua tarefa de efetuar a venda do software, fornecendo informações sobre a situação do cliente e, também, registrando a venda. A análise do cadastro do software é por ele efetuada em virtude de verificação de estoque.

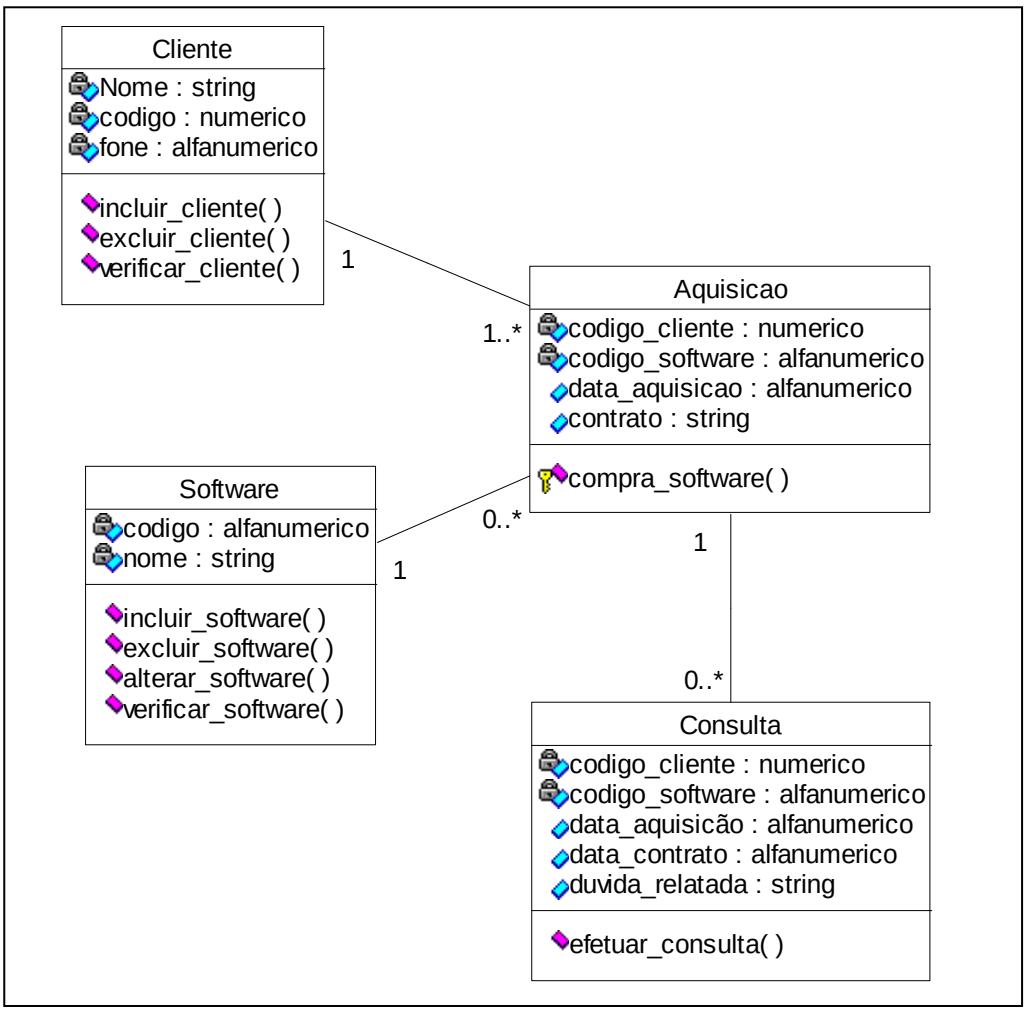
Na figura 8 tem-se ainda o funcionário do suporte, o qual tem por finalidade efetivar a consulta em relação às dúvidas sobre um determinado software relatadas por um cliente da empresa. Primeiramente, este funcionário verifica a situação do cliente (para confirmar se o mesmo está cadastrado como cliente da empresa e se adquiriu o software), após analisa o cadastro do software, buscando suas especificações técnicas para uma melhor resposta às dúvidas do cliente.

Figura 8: Diagrama de Casos de Uso



A figura 9 identifica o diagrama de classes do sistema hipotético, que serve para demonstrar de onde foram tiradas as seqüências de atividades a serem executadas nos diagramas de seqüências.

Figura 9: Diagrama de Classes



O diagrama de classes possui as seguintes definições:

a) cliente: pessoa ou empresa que adquire produtos na empresa. Nesta classe tem-se as operações: incluir cliente (cadastrar o novo cliente da empresa com os atributos código, nome e telefone) e excluir cliente (quando o mesmo não pretende mais adquirir produtos da empresa). A operação verificar cliente é usada pelo vendedor para funções cadastrais, e pelo suporte para verificação da situação do cliente – já explicado no diagrama de casos de uso;

b) software: discriminação de mercadoria destinada à venda. Cada software pode ou não ter em estoque. Nesta classe tem-se as seguintes operações: incluir software (cadastrar o novo software colocado à venda pela empresa com os atributos código e nome), alterar software (quando o mesmo sofre alterações em suas especificações descritivas) e excluir software (quando não se tem mais este produto destinado à

venda). A operação verificar software é usada pelo vendedor para constatar se a empresa tem este produto em estoque, e pelo suporte para averiguar as especificações técnicas do produto;

c) aquisição: evento pelo qual produtos são entregues ao cliente em troca de pagamento correspondente. Nesta classe tem-se a operação compra software (quando o cliente adquire o software comercializado pela empresa). Para fechamento da venda, o vendedor determina ao sistema o código do cliente, o código do software adquirido, a data de aquisição e o contrato de venda;
d) consulta: evento pelo qual o cliente relata dúvidas sobre o software adquirido e estas são dirimidas pelo suporte da empresa. Nesta classe tem-se a operação efetua consulta. Para efetivar a consulta, o suporte consulta no sistema o código do cliente, o código do software, a data de aquisição, a data de contrato e determina ao mesmo a dúvida relatada.

No diagrama, as classes de objetos associam-se da seguinte forma:

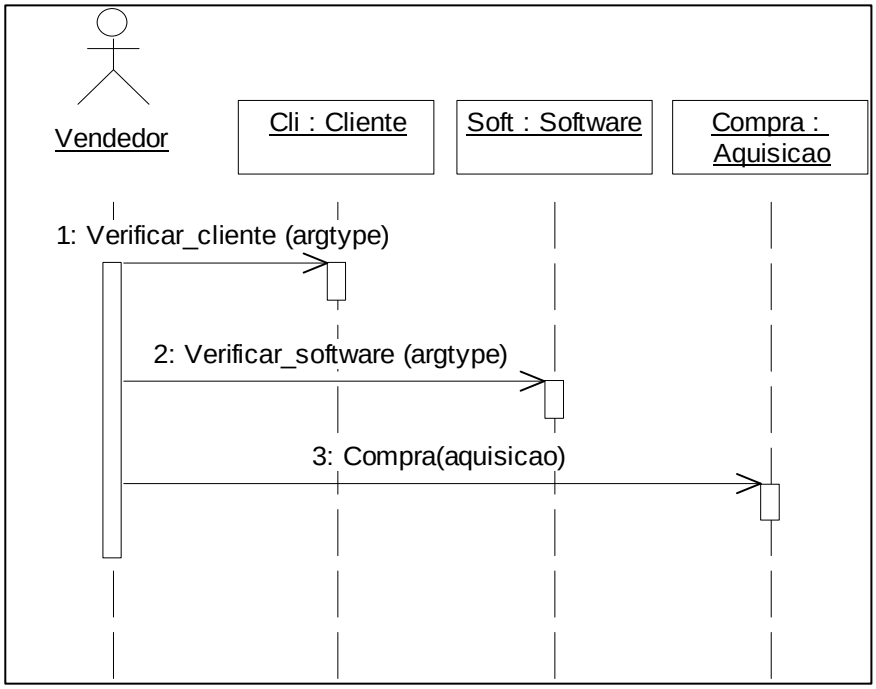
a) um cliente (classe cliente) pode adquirir um ou vários softwares (classe aquisição);
b) um tipo de software (classe software) pode ter várias cópias vendidas ou mesmo nenhuma (classe aquisição);
c) um tipo de software adquirido (classe software) pode ter várias dúvidas sobre ele consultadas pelo cliente ou nenhuma dúvida (classe consulta).

As classes podem ser entidades físicas como Cliente e Software ou abstratas como é o caso de Aquisição e Consulta. No entanto, todas elas denotam conceitos que fazem sentido ao domínio da aplicação.

O modelo serve apenas para oferecer uma visão geral das classes envolvidas e seus principais relacionamentos, orientando os passos seguintes da construção do software. Portanto, as classes aqui estabelecidas não estão em suas formas finais. Com o refinamento da análise podem ser acrescentados mais detalhes a estas classes, como a inclusão de novos atributos e operações e a definição de outras associações.

A seguir define-se os diagramas de seqüências, os quais mostram as seqüências de ações possíveis no sistema, conforme descrito nas figuras 10 e 11.

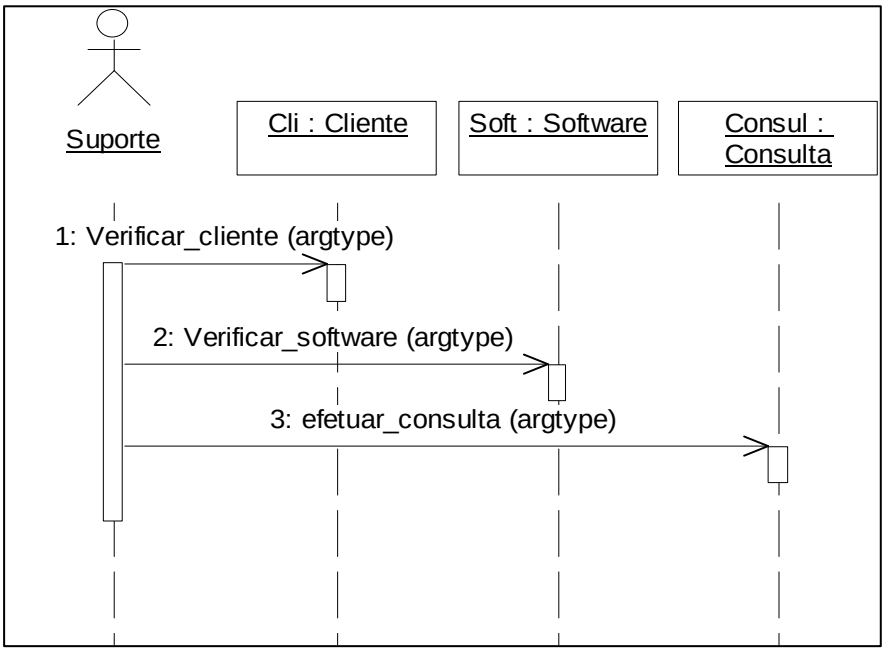
Figura 10: Diagrama de Seqüências para venda do software



O diagrama de seqüência apresenta uma interação típica entre usuário e sistema, organizada na escala do tempo. Geralmente, são necessários vários diagramas para explicar um caso de uso, um diagrama para cada tipo de interação possível. Por exemplo, para a ação verificar cliente observa-se o cadastro de clientes da empresa e pode-se incluir ou excluir clientes. Num sistema mais detalhado são utilizados dois diagramas de seqüências para representar as interações de inclusão e exclusão de clientes.

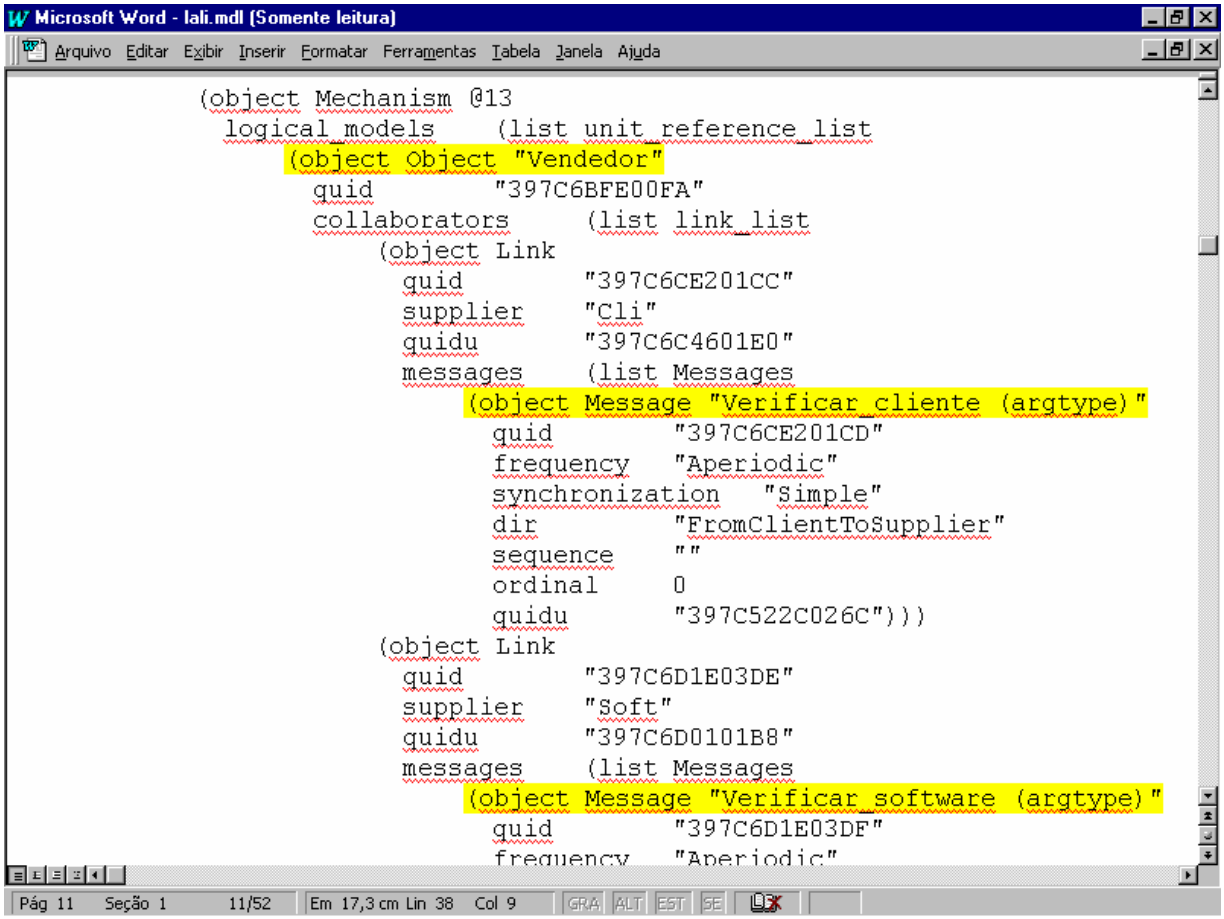
A figura 11 mostra o diagrama de seqüências para consulta do software pelo cliente.

Figura 11: Diagrama de Seqüências para consulta junto ao suporte pelo cliente



Os dados do diagrama de casos de uso, diagrama de classe e diagramas de seqüências são modelados na ferramenta Rational Rose. Ao salvar o arquivo de modelagem, a ferramenta gera um arquivo de extensão .mdl, onde estão inseridos todas as classes e operações dos objetos especificados pela modelagem de dados, conforme figura 12.

Figura 12: Arquivo com os parâmetros da modelagem



Nesta tela estão demonstrados parte dos exemplos de palavras chaves utilizadas pela modelagem para identificar nomes de classes (Objetos) e funções (mensagens).

As palavras-chaves destacadas no exemplo são do diagrama de seqüências de compra do software:

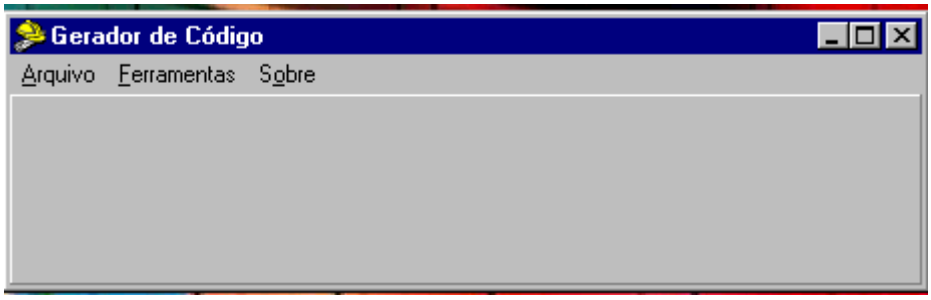
a) *Object* (com O maiúsculo) – indica o início de um objeto, e o nome do objeto, no caso Vendedor;

b) *Message* – indica uma mensagem. O que aparece entre parênteses nesta linha (dentro das aspas duplas) é a classe parâmetro para a mensagem. Para efeito do protótipo, vale lembrar que o procedimento interno do método associado e seu processamento é de responsabilidade do programador. A mensagem não passa de uma seqüência de ação a ser executada pelo sistema.

A partir deste arquivo .mdl gerado pela ferramenta CASE Rational Rose é que o protótipo realiza sua função de gerar o código fonte. Para isso, ele lê o arquivo e monta duas estruturas: uma passa pelo arquivo todo para retirar somente as classes e a outra faz nova leitura do arquivo para retirar as funções e guardar a que classe pertencem. Obtendo estas informações, o protótipo têm condições de gerar, mesmo que parcialmente, o código fonte, o qual contém as declarações das classes e das funções e também uma área de interface pronta para ser implementada.

O protótipo possui uma tela principal, a qual se divide em 3 menus, que são: Arquivo, Ferramentas e Sobre (este apenas uma abertura de tela sobre o que representa o protótipo – figura 13).

Figura 13: Tela principal do protótipo



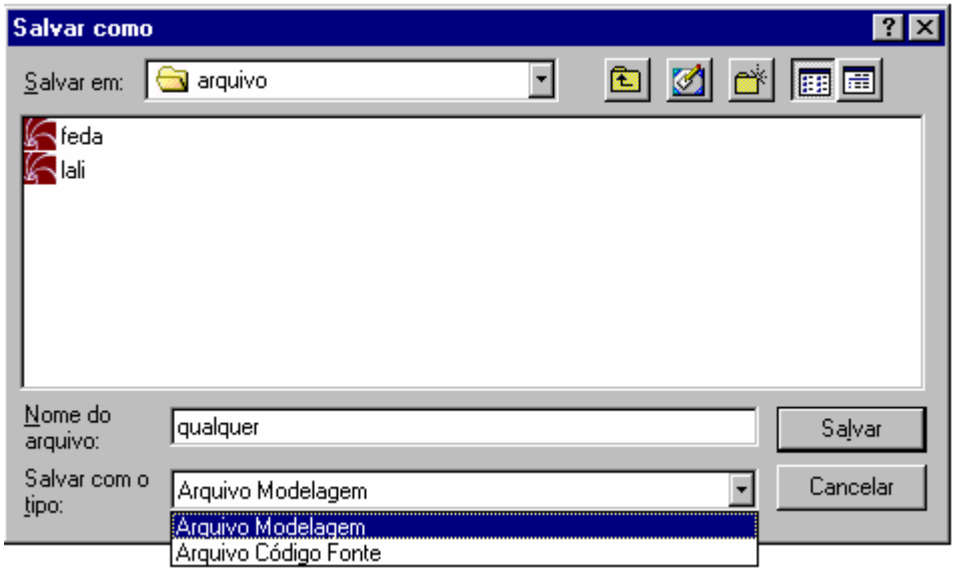
O menu Arquivo, representado na figura 14, é dividido em:

a) Abrir: abre o arquivo que o desenvolvedor desejar;
b) Fechar: fecha o arquivo que estava sendo manipulado pelo desenvolvedor;
c) Salvar: salva o arquivo;
d) Sair: sai do sistema.

Figura 14: janela do menu arquivo

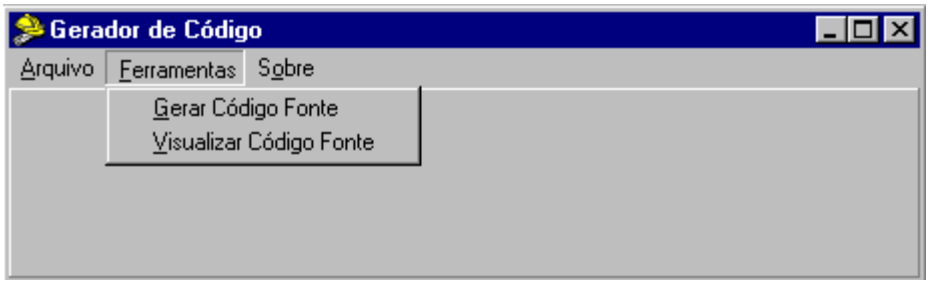
A figura 15 identifica o processo que salva um arquivo de modelagem do protótipo ou um arquivo cujo código fonte já tenha sido gerado.

Figura 15: tela para salvar arquivo do protótipo



A figura 16 mostra o menu ferramentas a ser utilizado para a geração do código fonte em relação ao arquivo especificado.

Figura 16: menu Ferramentas

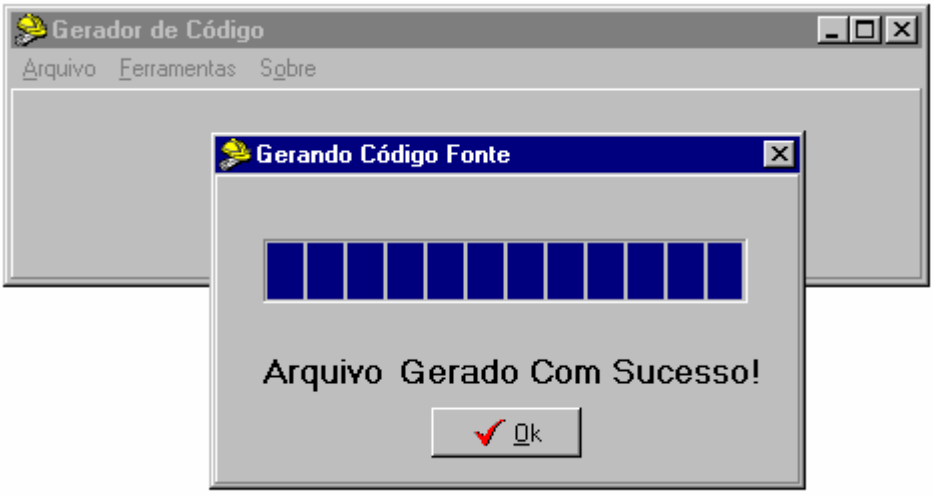


O menu Ferramentas é dividido em:

a) Gerar Código Fonte: inicia o processo de geração do código contendo as classes e objetos do sistema;
b) Visualizar Código Fonte: permite a visualização do conteúdo gerado como código fonte.

A figura 17 mostra a tela de execução com a geração de um código fonte.

Figura 17: arquivo com código fonte gerado – Tela de Execução



Na figura 17 o sistema está lendo o arquivo origem, e processando suas linhas. Como visto, o arquivo gerado pelo Rational Rose pode ser lido sem problemas por qualquer editor, inclusive o Microsoft Word. Sendo assim, pode ser lido linha a linha. O protótipo procura por algumas “palavras chaves” que identificam nome do arquivo, início e final de um bloco referente a uma classe, assim como a mensagem.

A primeira coisa que o protótipo procura é pelo nome do arquivo. Encontrado este nome o protótipo cria o arquivo fonte, e posiciona o arquivo origem (.mdl) de novo no início. Neste ponto o protótipo busca pela palavra chave que indica o início de um bloco referente a uma classe.

Após identificado o início e o final desta classe, são identificadas as mensagens (funções ou seqüências de ações) da classe e é verificado se os parâmetros destas mensagens já foram identificados, o que significaria já ter sido processado pelo protótipo, não havendo, portanto, necessidade de nova identificação.

Lembra-se que os parâmetros das mensagens, pelo nível de abstração, são outras classes, e no ambiente de desenvolvimento Delphi as classes devem obedecer uma ordem de

declaração. Não se pode utilizar uma classe como parâmetro antes dela ser declarada. A classe(objeto) somente é processada quando todas as classes utilizadas pelas suas mensagens já tiverem sido declaradas.

Neste processo o arquivo origem é lido tantas vezes quantas forem necessárias, sendo que o protótipo, através de uma estrutura de variáveis estáticas e dinâmicas de memória, possui um controle para guardar as linhas dos objetos ainda não processadas.

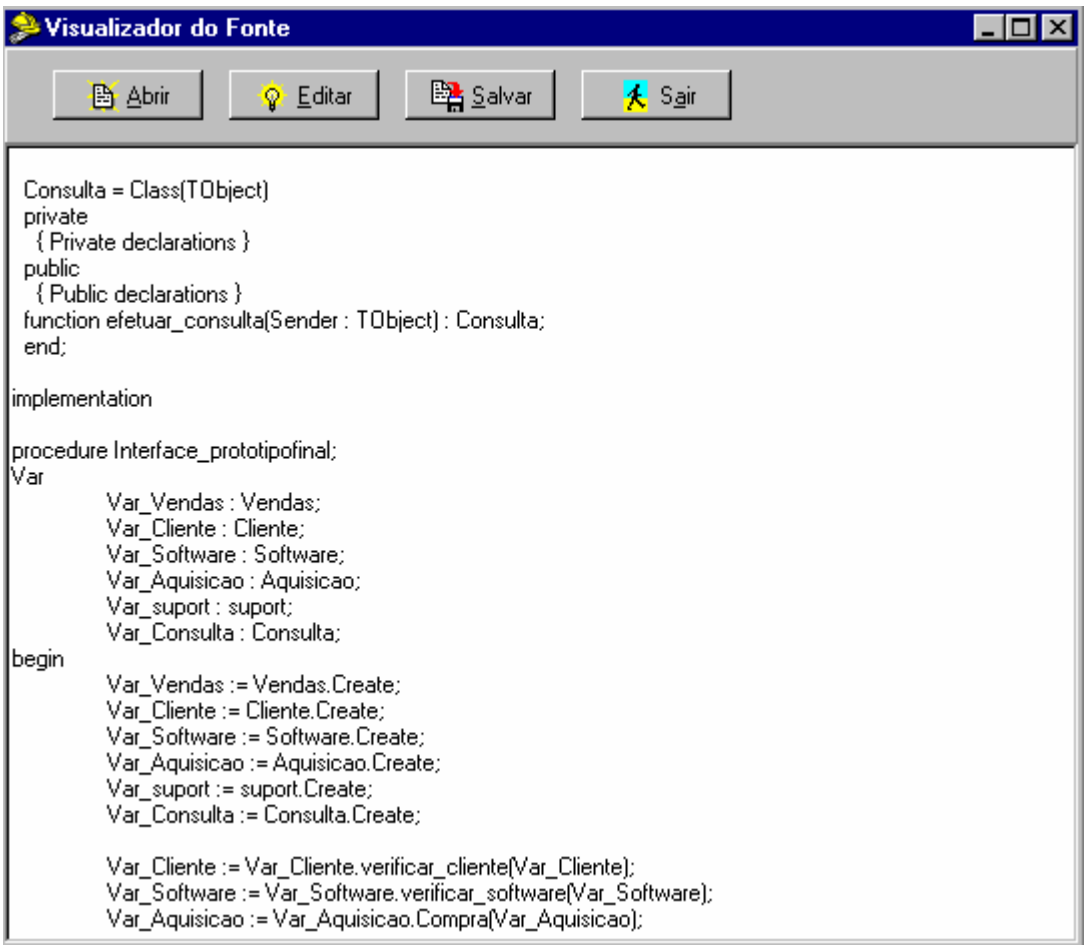
Ao final de todo este processo o arquivo fonte é gerado com uma ordem lógica que não causa nenhum problema quando compilado pelo ambiente Delphi.

Importante salientar que a tela visualizada na figura 17 mostra graficamente o progresso do processamento do arquivo fonte, e que a indicação de que o arquivo fonte foi gerado com sucesso confirma que não foi encontrado nenhum problema que impedisse a geração do mesmo. Após a geração é chamada automaticamente a tela de visualização do arquivo fonte gerado para que o usuário possa ver o resultado do processamento do arquivo do Rational Rose.

Se precisar visualizar o código fonte gerado ou mesmo editar um código fonte já existente pode-se fazê-lo na tela demonstrada pela figura 18, com o seguinte menu:

a) Abrir: abre um arquivo com código fonte gerado;
b) Editar: serve para efetuar alterações de implementação no código fonte. Lembrar sempre que as alterações efetuadas devem manter consistência com a modelagem de dados. O comando editar chama o ambiente Delphi (ferramenta própria de utilização para este fim), para efeito de compilação e verificação de erros;
c) Salvar: salva o arquivo editado;
d) Sair: sai da tela de visualização de código fonte.

Figura 18: Tela de visualização e edição do código fonte com o código fonte gerado



A tela apresenta o código fonte gerado mostrando as classes existentes para implementação e as funções pertencentes às classes. No anexo 1, o arquivo completo do código fonte gerado pelo protótipo, onde as informações sobre as classes foram retiradas do arquivo utilizado como origem, que é o arquivo .mdl gerado pela ferramenta Rational Rose. O Var indica variável e como o ambiente Delphi não aceita uma variável com o mesmo nome do tipo (ou do objeto), criou-se da seguinte forma: Var_Cliente p/ o objeto Cliente, seguindo a mesma lógica para os demais. Já as palavras reservadas (private, public, begin, end, etc.) do ambiente Delphi foram pesquisadas no próprio Delphi e são encaixadas no arquivo fonte em ordem lógica indicada pelo arquivo origem processado pelo protótipo.

As classes são declaradas como sendo do tipo Tobject por ser o tipo mais genérico e abstrato de objeto que o ambiente Delphi proporciona. Outra justificativa é que as classes são

declaradas como objetos no diagrama de seqüência, o que faz a escolha do Tobject uma decisão mais lógica.

No anexo 1, podemos verificar que as funções são declaradas passando o Sender e retornando a própria classe. Isto é feito porque não se sabe o que será acrescentado à classe pelo usuário em termos de atributos e subfunções, fazendo-se necessário que a troca de informação entre as funções seja a mais abstrata possível.

O protótipo não se destina a gerar executáveis e sim arquivos fonte “BASE” para serem utilizados pelo ambiente Delphi, logo ele não se propõe a compilar ou identificar eventuais erros que o fonte possa ter, como por exemplo uma má definição do diagrama de seqüência. Para compilar, deve-se chamar o ambiente Delphi através da tecla editar do visualizador do fonte.

Após o arquivo processado pelo protótipo é gerado o arquivo fonte, extensão .pas e após este arquivo ser gravado, é gerado o arquivo “Aplicacao_” + Nome_Arquivo_Fonte ou arquivo .dpr (que indica uma aplicação do Delphi – figura 21). Este arquivo é o arquivo passado como parâmetro para o ambiente Delphi (figura 22). Deve então o usuário abrir o arquivo fonte .pas para poder compilá-lo (figura 23). Isso foi feito porque o ambiente Delphi somente compilaria o fonte se este estivesse associado a uma aplicação ou um pacote de funções (*packages*). Do modo como foi optado o fonte está pronto para ser utilizado.

Isso somente ocorre quando o arquivo Origem é processado e chega-se à tela de visualização. Se não tiver processado o arquivo Origem e entrar na tela de visualização, abrir um arquivo (com o botão abrir) e então clicar no botão editar, o Delphi abrirá somente este arquivo, sem a aplicação, pois não há como identificar a que aplicação o arquivo aberto pertence.

As figuras 19, 20, 21, 22 e 23 mostram a seqüência de criação dos arquivos gerados pelo protótipo e a compilação no ambiente de desenvolvimento Delphi.

Figura 19: Arquivo lali.mdl criado e disponível para uso

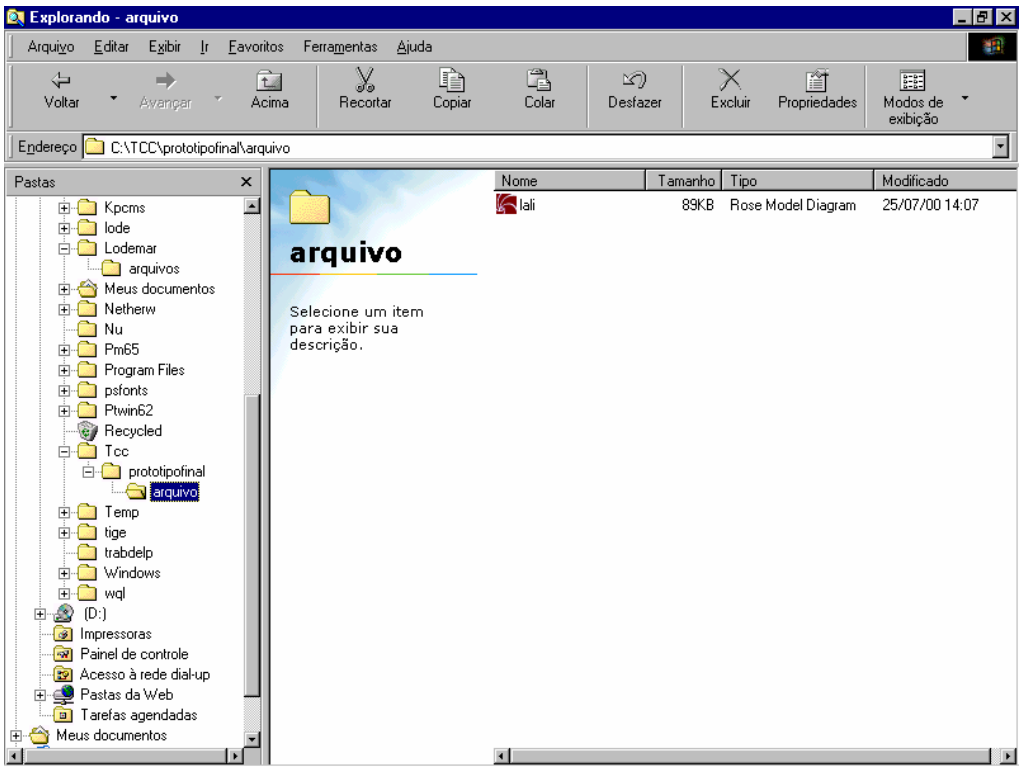


Figura 20: O arquivo lali.mdl sendo aberto pelo protótipo

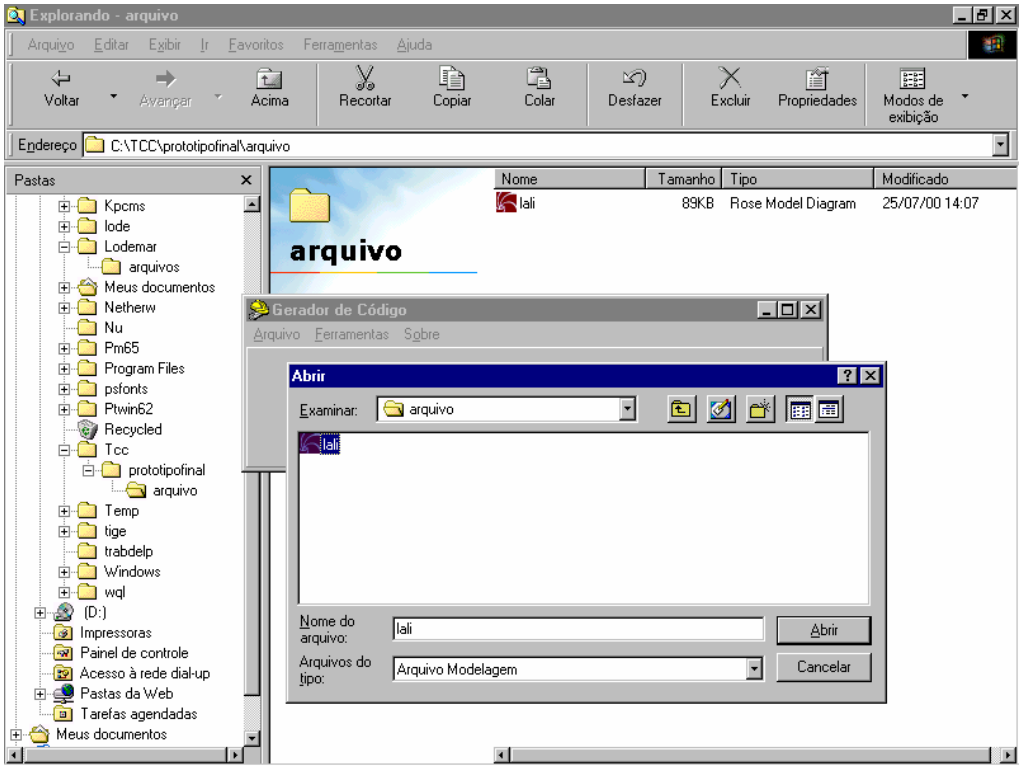


Figura 21: Arquivos gerados pelo protótipo na geração do código fonte

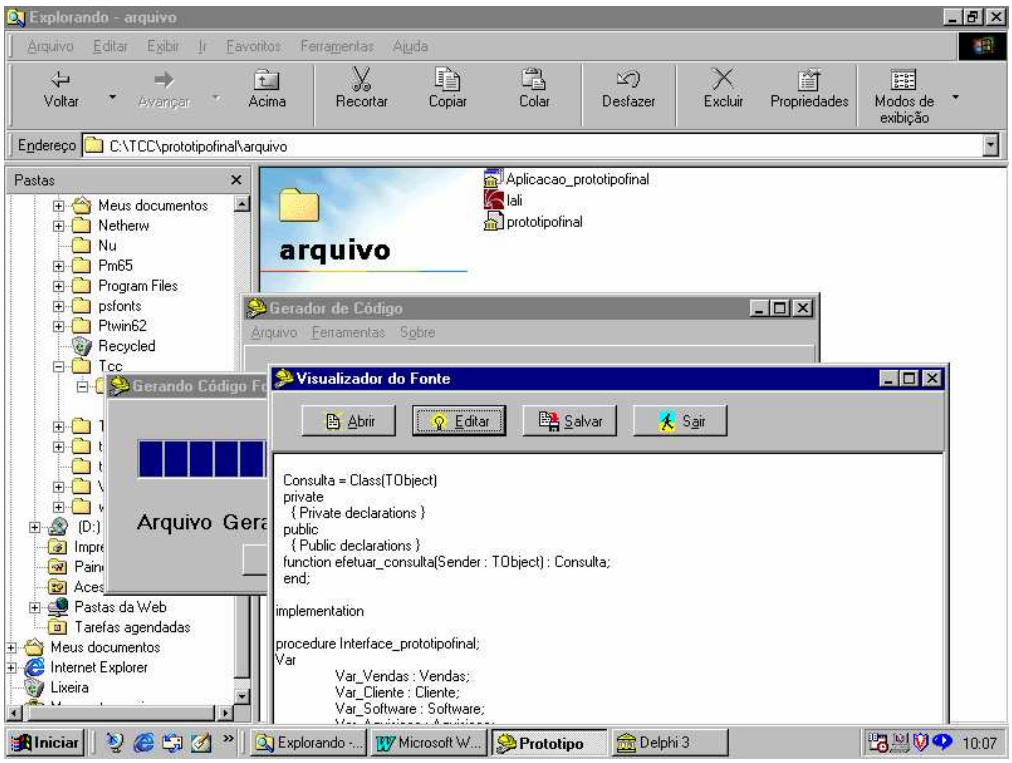


Figura 22: Arquivo que é aberto quando se solicita edição do código fonte. O próprio protótipo chama o Delphi e abre este arquivo

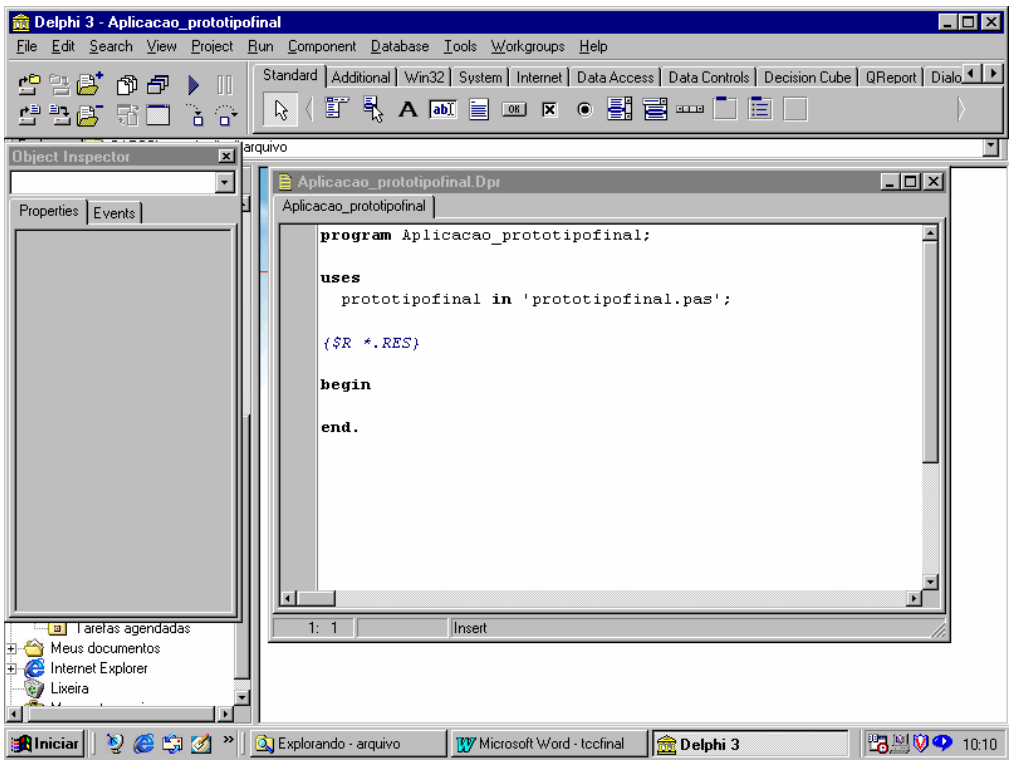
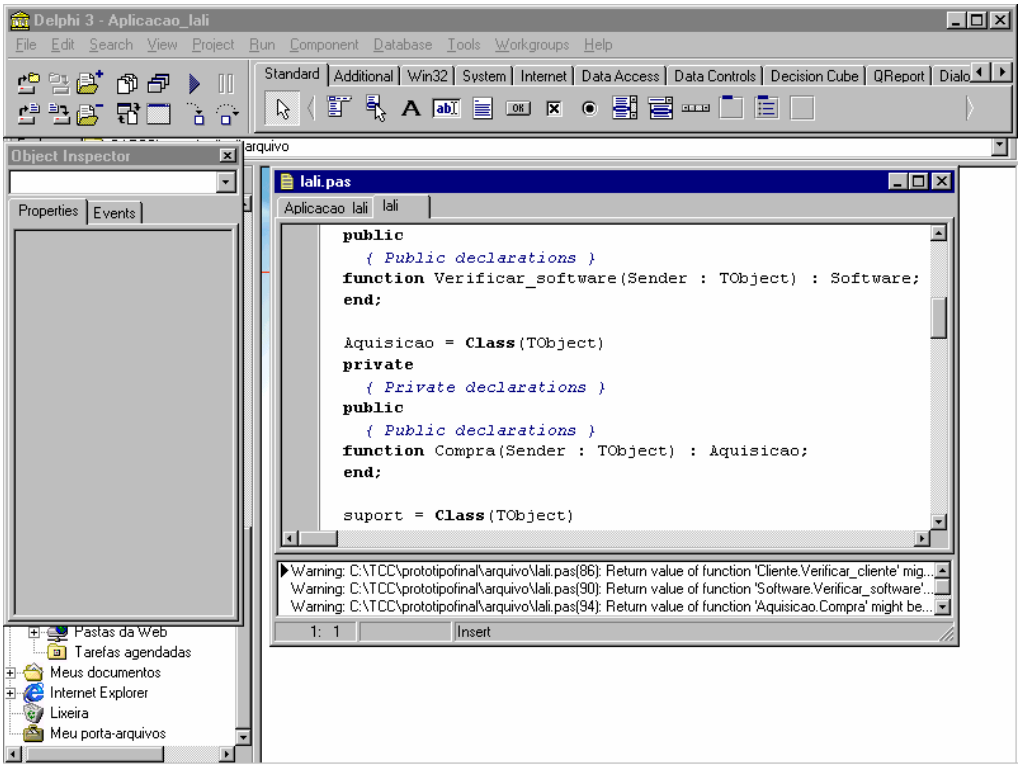


Figura 23: Arquivo .pas do código fonte compilado



Neste caso, após a compilação no ambiente Delphi requer-se apenas retorno de valores de função, parâmetros estes que serão definidos pelo desenvolvedor que irá implementar o código fonte gerado, caso já não tenham sido definidos na ferramenta CASE.

6 CONCLUSÕES

A criação de uma linguagem para a comunidade de desenvolvedores em orientação a objetos era uma necessidade antiga. A UML realmente incorporou muitos recursos que dão à linguagem uma extensibilidade muito grande.

Sem dúvida a linguagem UML facilitará às empresas de desenvolvimento de software uma maior comunicação e aproveitamento dos modelos desenvolvidos pelos seus vários analistas envolvidos no processo de produção de software já que a linguagem que será utilizada por todos será a mesma, acabando assim com qualquer problema de interpretação e mal-entendimento de modelos criados por outros desenvolvedores. Os modelos criados hoje poderão ser facilmente analisados por futuras gerações de desenvolvedores acabando com a diversidade de tipos de nomenclaturas de modelos, o grande empecilho do desenvolvimento de softwares orientados a objetos.

A UML constitui uma linguagem para especificação de sistemas a partir de uma série de recursos a serem aplicados durante todos os passos do desenvolvimento. Ela estabelece uma rica semântica e uma minuciosa notação, cobrindo pontos fracos de outras técnicas. No entanto, é extensa e complexa, além de exigir mudança de paradigmas para a compreensão da maioria dos seus itens. Oferece tantos instrumentos ao desenvolvedor que fica difícil definir o que é fundamental, o que é opcional e até mesmo quando utilizar alguns deles.

Em relação ao protótipo, conseguiu-se obter parcialmente código fonte a partir de diagramas modelados no Rational Rose. Quando bem organizada, a modelagem permite a localização fácil e rápida das classes envolvidas em determinada funcionalidade, não havendo dificuldades para a incorporação das novas características.

Durante a coleta de informações foram pesquisadas várias fontes, entre elas livros, revistas, publicações e a Internet, que ofereceu a oportunidade da observação do nível das questões debatidas em fóruns de discussão. Notou-se que várias pessoas se deparavam com as mesmas dúvidas em relação ao desenvolvimento de sistemas orientados a objetos e,

principalmente, sobre a reutilização de códigos fontes. Isto motivou ainda mais o estudo para a busca de soluções.

Dentre as dificuldades encontradas, destaca-se a falta de uma melhor conceituação em termos de orientação a objetos e UML, o que limitou o protótipo a gerar parcialmente o código fonte de alguns arquivos modelados na ferramenta CASE Rational Rose, embora outros tenham sido gerados de forma completa.

Sugere-se como extensão ao trabalho gerar diagramas de seqüências dentro do próprio ambiente de desenvolvimento Delphi, facilitando assim o trabalho do desenvolvedor, que poderia cada vez mais condensar-se em uma única ferramenta.

ANEXO 1 – CÓDIGO FONTE GERADO

```
unit prototipofinal;

interface

type

  Vendas = Class(TObject)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

  Cliente = Class(TObject)
  private
    { Private declarations }
  public
    { Public declarations }
  function verificar_cliente(Sender : TObject) : Cliente;
  end;

  Software = Class(TObject)
  private
    { Private declarations }
  public
    { Public declarations }
  function verificar_software(Sender : TObject) : Software;
  end;

  Aquisicao = Class(TObject)
  private
    { Private declarations }
  public
    { Public declarations }
  function Compra(Sender : TObject) : Aquisicao;
  end;
```

```
  suport = Class(TObject)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

  Consulta = Class(TObject)
  private
    { Private declarations }
  public
    { Public declarations }
  function efetuar_consulta(Sender : TObject) : Consulta;
  end;

implementation

procedure Interface_prototipofinal;
Var
        Var_Vendas : Vendas;
        Var_Cliente : Cliente;
        Var_Software : Software;
        Var_Aquisicao : Aquisicao;
        Var_suport : suport;
        Var_Consulta : Consulta;
begin
        Var_Vendas := Vendas.Create;
        Var_Cliente := Cliente.Create;
        Var_Software := Software.Create;
        Var_Aquisicao := Aquisicao.Create;
        Var_suport := suport.Create;
        Var_Consulta := Consulta.Create;

        Var_Cliente := Var_Cliente.verificar_cliente(Var_Cliente);
        Var_Software := Var_Software.verificar_software(Var_Software);
        Var_Aquisicao := Var_Aquisicao.Compra(Var_Aquisicao);
        Var_Consulta := Var_Consulta.efetuar_consulta(Var_Consulta);
```

```
        Var_Vendas.Destroy;
        Var_Cliente.Destroy;
        Var_Software.Destroy;
        Var_Aquisicao.Destroy;
        Var_suport.Destroy;
        Var_Consulta.Destroy;
end;

function Cliente.verificar_cliente(Sender : TObject) : Cliente;
begin
end;

function Software.verificar_software(Sender : TObject) : Software;
begin
end;

function Aquisicao.Compra(Sender : TObject) : Aquisicao;
begin
end;

function Consulta.efetuar_consulta(Sender : TObject) : Consulta;
begin
end;

end.
```

REFERÊNCIAS BIBLIOGRÁFICAS

[BAR1997] BARBIERI, Carlos. A unificação dos métodos de orientação a objetos. **Revista Developers Magazine**. Rio de Janeiro, n.12, p. 20-24, ago. 1997.

[BAR1998] BARROS, Pablo Fernando do Rêgo. **UML – Linguagem de Modelagem Unificada**, 1998. Endereço eletrônico : http://www.eribeiro.com.br/pablo/uml/index.html. Data da consulta : 14/06/2000

[CAN1998] CANTÚ, Marco. **Delphi 3 : a bíblia do programador**. São Paulo : Makron Books, 1998.

[COA1991] COAD, Peter & YOURDON, Ed. **Análise baseada em objetos**. Rio de Janeiro : Ed. 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**. Nova York, EUA : Ed. Wiley, 1998.

[FOW1997] FOWLER, Martin. **UML Distilled**. Canadá : Addison-Wesley, 1997.

[FUR1998] FURLAN, José Davi. **Modelagem de objetos através da UML**. Rio de Janeiro : Makron Books, 1998.

[JUN1998] JÚNIOR, Silvino Schlickmann. **Linguagens orientadas a objetos**, 1998. Endereço eletrônico : http://www.cfh.ufsc.br/~junior/linguagens.htm. Data da consulta : 16/06/2000

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

[MUL1997] MULLER, Pierre-Alain. **Instant UML**. Canadá : Wrox Press Ltd., 1997.

[PRE1998] PREBIANCA, Jefferson Luiz. **Protótipo de uma ferramenta CASE para especificação do modelo funcional segundo a proposta UML**. Blumenau, 1998. Monografia (Bacharelado em Ciências da Computação) Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau.

[RAT1997] RATIONAL, Software Corporation. **Unified modeling language**, version 1.2 1998. Endereço eletrônico : http:/www.rational.com/uml. Data da consulta: 15/04/2000.

[ROC1996] ROCHA, Helder L. S. da. **Tutorial – programação orientada a objetos**, 1996. Endereço eletrônico : http://www.dsc.ufpb.br/~helder/java/Cap_2.html. Data da consulta : 10/06/2000.

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

[SCO1996] SCOLA, Antônio Carlos. Orientação a objetos uma saída para a crise do software. **Revista Developers Magazine**, Rio de Janeiro, n. 4, p. 16-17, dez. 1996.