TABA OBJ: UM AMBIENTE DE DESENVOLVIMENTO
DE SOFTWARE COM ORIENTACAO A OBJETOS

Adriana Lima de Queirós Mattoso

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇAO DOS PROGRAMAS DE PÓS-GRADUAÇAO EM ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSARIOS PARA A ORIENTAÇÀO DO GRAU DE MESTRE EM CIÉNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇAO.

Aprovada por:

Profa. Ana Regina C. da Rocha, D. Sc.
(Presidente)

Profa. Sonia S. Sette, D. Ing.

Prof. Jano Moreira de Souza, PhD.

MATTOSO, ADRIANA LIMA DE QUEIRÓS
TABA_OBJ: Um Ambiente de Desenvolvimento de Software com Orientacão a Objetos [Rio de Janeiro] 1990
XI, 148 p. 29,7cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computacão, 1990)
Tese - Universidade Federal do Rio de Janeiro, COPPE
1. Ambientes de Desenvolvimento de Software
I. COPPE/UFRJ II. Título (série).

Ao Luiz e à Lúcia

Agradecimentos

Ao Luiz Carlos pela sua dedicação e apoio constantes.

À Lúcia pelo seu carinho.

Aos meus pais pela motivação e apoio.

À minha irmã Marta pela sua ajuda sempre que necessária.

À minha irmã Cecília pelo seu apoio.

À Professora Ana Regina Cavalcanti da Rocha pela oportunidade deste tema de tese e pela dedicação e orientação deste trabalho.

Aos professores Sonia Schechtman Sette e Jano Moreira de Souza por darem a honra de participarem da minha banca de tese.

Ao Guilherme Horta Travassos pelo apoio sempre que necessário.

Ao colega Geraldo Xexeo por me ajudar a avaliar o método.

À colega Maria Cristina Passos.

Aos professores e funcionários da COPPE/Sistemas.

À CAPES pelo apoio financeiro.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para obtenção do grau de Mestre em Ciências (M. Sc.).

TABA_OBJ: Um Ambiente de Desenvolvimento de Software com Orientação a Objetos

Adriana Lima de Queirós Mattoso

Agosto de 1990

Orientador: Ana Regina Cavalcanti da Rocha, D. Sc.

Programa: Engenharia de Sistemas e Computação

Este trabalho apresenta o TABA_OBJ, um ambiente de desenvolvimento de software com orientação a objetos, para a Estação TABA. Este ambiente visa apoiar o processo de desenvolvimento de software baseado no paradigma de objetos, oferecendo um método, um modelo de ciclo de vida e instrumentos que auxiliam durante as etapas do desenvolvimento. Finalmente é apresentado o Editor de Diagrama de Classes, uma ferramenta gráfica de apoio à fase de Análise do método proposto no TABA_OBJ.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfillment of the requirements for the degree of Master of Science (M. Sc.).

TABA_OBJ: An Object-Oriented Software Development Environment

Adriana Lima de Queirós Mattoso

August, 1990

Thesis Supervisor: Ana Regina Cavalcanti da Rocha, D. Sc.

Department: Computer Systems Engineering

This work presents TABA_OBJ, an Object-Oriented Software Development Environment, for the TABA Work Station. This environment supports the development process based on the object paradigm, providing a method, a life-cycle model and instruments that aids the development phases. Finally, the Class Diagram Editor is presented, a supporting graphic tool for the analysis phase of the TABA_OBJ method.

INDICE

III. TABA_OBJ: UM AMBIENTE DE DESENVOLVIMENTO DE SOFTWARE COM ORIENTACAO A OBJETOS

INDICE DE FIGURAS

INDICE DE TABELAS

CAPÍTULO I - INTRODUÇÃO

Este capítulo descreve os fatores que motivaram o desenvolvimento desta tese (seção I.1), define seus objetivos (seção I.2) e apresenta sua organização (seção I.3).

I.1 - Motivação

Durante os anos 80, o paradigma de objetos ocupou posição de destaque em diversas áreas de pesquisa da informática. Tendo sua origem nas linguagens de programação como SIMULA e Smalltalk [1], a orientação a objetos provocou interesse em pesquisadores de áreas como Banco de Dados, Interface com o Usuário, Inteligência Artificial e Engenharia de Software. Segundo MONTE [2], a popularização do termo orientação a objetos se deve ao sucesso de alguns sistemas que humanizaram o uso e a programação de computadores. Esses sistemas, que foram autodefinidos como "orientados a objetos", são baseados em noções, consideradas intuitivas ao ser humano, tais como objetos, classes, estado, ação e reação.

A partir do Smalltalk, que foi a primeira linguagem de programação a trabalhar com essas noções, surgiram inúmeras linguagens e extensões de linguagens de programação tradicionais para suportar os elementos da orientação a objetos.

Com a difusão da programação orientada a objetos, sentiu-se a necessidade de se utilizar a filosofia da orientação a objetos em todo o processo de desenvolvimento, desde a especificação até a implementação e não somente na fase de implementação. Os métodos de desenvolvimento de software tradicionais, entretanto, organizam o projeto em torno de suas funções ou em torno de seus dados, diferentemente da programação orientada a objetos onde os

dados e todas as operações que manipulam esses dados, estão agrupados em uma única estrutura que são os objetos.

Caso desde o início do desenvolvimento se utilizasse a orientação a objetos, o desenvolvimento de software seria mais fácil, pois se trabalharia sempre com o mesmo elemento de abstração, os objetos. Apesar de existirem diversas propostas para desenvolvimento de software com orientação a objetos [3], [4], [5], [6] e [7], não há ainda um método já sedimentado que cubra todo o processo de desenvolvimento.

Houve, então, uma grande motivação de se estudar o paradigma da orientação a objetos e sua aplicação na Engenharia de Software. Todavia, como não encontrou-se nenhum estudo propondo um ambiente de desenvolvimento de software, com um método, um modelo de ciclo de vida e instrumentos e ferramentas que dessem suporte ao desenvolvimento de software com orientação a objetos, sentiu-se motivação para realizar estudos de maneira a propor um ambiente de desenvolvimento de software orientado a objetos.

Por outro lado, está em andamento na COPPE/UFRJ o Projeto TABA [8], que visa a construção de uma estação de trabalho, configurável para desenvolvimento de software que suporta desde a criação de ambientes de desenvolvimento de software que consideram as características dos sistemas a serem desenvolvidos até a própria execução destes sistemas.

Deste modo, sentiu-se a motivação de propor um ambiente de desenvolvimento de software orientado a objetos que possa fazer parte da Estação TABA, o ambiente TABA_OBJ.

I.2 - Objetivos da tese

O objetivo principal desta tese é a definição de um ambiente de desenvolvimento de software orientado a objetos para a estação TABA, o ambiente TABA_OBJ.

Para atingir esse objetivo, foi feita inicialmente, uma revisão bibliográfica, definindo os conceitos básicos da orientação a objetos, descrevendo algumas linguagens de programação orientadas a objetos e apresentando os métodos de desenvolvimento de software com orientação a objetos existentes na literatura.

Posteriormente, foi proposto um método de desenvolvimento de software, cobrindo todo o processo de desenvolvimento. A elaborção deste método foi feita em três etapas. Inicialmente, foi proposto um método [9] que teve como objetivo ser mais abrangente, ao contrário dos métodos encontrados na literatura que cobriam somente a fase de projeto ou de análise. Após ter-se utilizado o método em dois projetos, a proposta foi avaliada e sofreu algumas modificações. A partir desta experiência, o método foi refeito e apresentado em [10].

Para a elaboração final do método, foi feito um estudo crítico dos modelos de ciclo de vida e roteiros de documentação pesquisados. Finalmente, foi implementada uma das ferramentas propostas para o TABA_OBJ, utilizando-se o ambiente de programação Actor [11].

I.3 - Organização da tese

Esta tese está organizada em cinco capítulos. O capítulo I contém a introdução e apresenta os fatores que motivaram sua elaboração, define os objetivos da tese e sua organização.

O capítulo II faz uma revisão bibliográfica da orientação a objetos, definindo seus conceitos básicos, apresentando linguagens de programação e métodos para o desenvolvimento de software com orientação a objetos.

O capítulo III trata de ambientes de desenvolvimento de software orientado a objetos. Justifica-se a definição do ambiente TABA_OBJ e apresenta-se a definição de termos

de ambientes de desenvolvimento de software utilizada nesta tese. Ainda neste capítulo, são feitos estudos sobre modelos de ciclo de vida e roteiros de documentação. Finalmente é apresentado o método proposto, as críticas feitas a partir de sua utilização e a nova versão do método.

No capítulo IV apresenta-se a especificação e projeto da ferramenta, bem como uma descrição de sua implementação.

No capítulo V são expostas as conclusões da tese e o capítulo VI contém as referências bibliográficas.

CAPÍTULO II - ORIENTAÇÃO A OBJETOS: UMA VISÃO GERAL

Este capítulo apresenta uma visão geral da filosofia e dos conceitos básicos da orientação a objetos e descreve métodos de desenvolvimento de software e linguagens de programação que usam o paradigma de objetos.

Na seção II.1 serão apresentados os conceitos básicos e características da orientação a objetos, na seção II.2 serão mostradas algumas linguagens de programação com orientação a objetos. Finalmente, a seção II.3 apresentará alguns métodos de desenvolvimento de software orientado a objetos.

II.1 - Conceitos Básicos

A filosofia da orientação a objetos se baseia na noção intuitiva que as pessoas têm de objeto, i.e., que o mundo é povoado por objetos que interagem entre si. A idéia, portanto, é pensar em desenvolvimento de software composto por objetos e não por *dados*, *procedimentos* e *controle* como elementos isoladamente.

Nos métodos de desenvolvimento de software tradicionais há uma distância muito grande entre o problema no mundo real e sua abstração no produto final ("gap semântico"). A idéia da orientação a objetos é de se trabalhar com noções intuitivas (objetos e ações) durante todo o ciclo de desenvolvimento, atrasando-se ao máximo a introdução de conceitos de processamento de dados, diminuindo portanto essa distância.

O paradigma de orientação a objetos teve origem nas linguagens de programação como SIMULA (1967), LISP e Smalltalk (1968). Porém, Smalltalk foi a primeira linguagem que transformou o paradigma de objetos num modelo concreto. Após inúmeras pesquisas no laboratório PARC da Xerox, prosseguindo o projeto de Alan Kay, GOLDBERG [1] conseguiu,

com a construção do Smalltalk, definir claramente os conceitos básicos da orientação a objetos. Apresentaremos, a seguir, as definições desses conceitos.

II.1.1 - Definições [1], [12], [13] e [14]

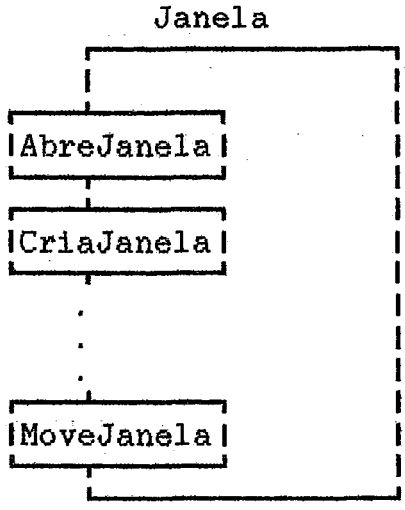

Figura 1 - Uma representação gráfica do objeto Janela

Objeto

Entidade que consiste de dados e todas as operações que podem manipular esses dados. O único acesso aos dados desse objeto é através de suas operações. Cada objeto reside na sua própria máquina abstrata e objetos se comunicam entre si através de mensagens [12]. Por exemplo, num sistema cuja interface seja baseada em janelas e cardápios, um objeto típico seria o objeto **Janela**. Este objeto agruparia todas as características comuns às janelas do sistema e também suas operações, que poderiam ser: AbreJanela, CriaJanela, FechaJanela, MoveJanela (Figura 1). Deste modo, todas as informações pertinentes a um objeto (seus atributos e operações) estariam concentradas (encapsuladas) neste objeto.

Classe

Conjunto de objetos com características semelhantes e que respondem às mensagens da mesma maneira. Um objeto é dito ser uma instância de classe. No caso do exemplo anterior, poderíamos pensar que o sistema poderia implementar várias janelas, todas possuindo um conjunto de características e operações em comum (Figura 2). Poderíamos pensar em reunir todas essas características em comum em uma única classe **Janela**, à que todas as janelas pertenceriam.

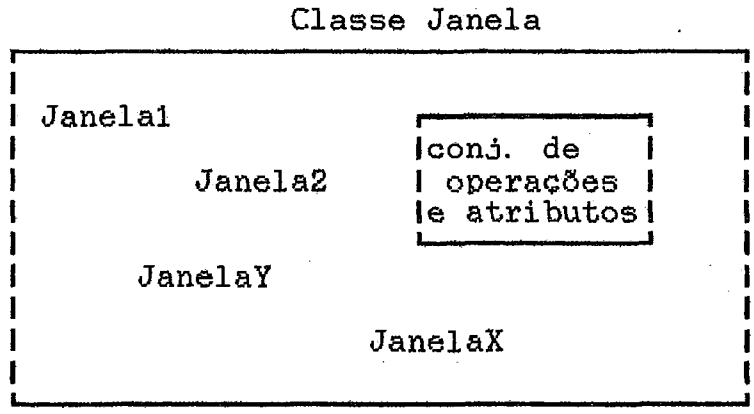

Figura 2 - Classe Janela

Mensagem

Solicitação de um objeto para que outro objeto efetue uma de suas operações. O objeto receptor pode enviar uma mensagem de volta com o valor desejado e também pode enviar mensagens para outros objetos e assim por diante [13].

Qualquer objeto do sistema que quisesse criar uma janela, por exemplo, enviaria uma mensagem de CriaJanela para o objeto Janela. Este objeto verificaria se ele sabe tratar essa mensagem, i.e., se ele possui a operação solicitada e, caso positivo, ele efetuaria esta operação. A operação de CriaJanela, poderia necessitar de uma função do tipo DesenhaCardápio, provocando o envio de uma outra

mensagem ao objeto Cardápio e assim por diante, como na Figura 3.

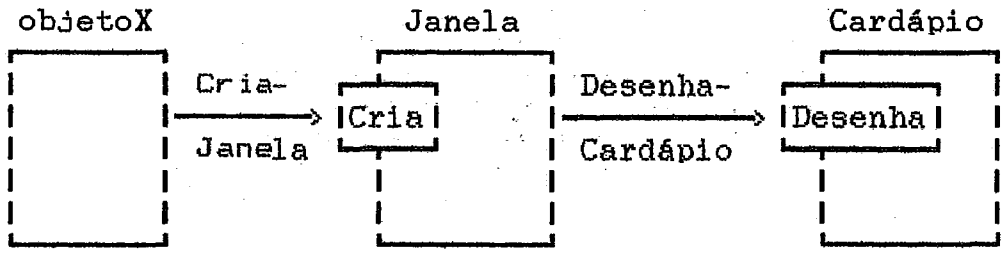

Figura 3 - Envio de mensagens entre objetos

Encapsulamento de Dados

Técnica de esconder a estrutura de um objeto atrás de um conjunto de operações sobre ele, sendo acessível somente através destas operações.

Seguindo o exemplo anterior, a única maneira de se comunicar com o objeto Janela é enviando-lhe uma mensagem. Caso se queira criar uma janela, não se tem acesso ao código interno da operação CriaJanela, isto é, só se comunica com um objeto através de sua interface, sendo sua parte interna inacessível.

Abstração de Dados

Utilização dos dados sem se preocupar com a sua estruturação interna.

Método

É o nome dado às operações de um objeto. Ao enviar uma mensagem para um objeto, o outro objeto (emissor da mensagem), está solicitando que o objeto receptor execute o respectivo método. Outros objetos não têm acesso ao código interno dos métodos de um objeto. Diversos objetos podem ter métodos diferentes, porém com o mesmo nome.

Usando o exemplo anterior, CriaJanela e DesenhaCardápio seriam métodos de Janela e Cardápio respectivamente.

Herança

Criação de subclasses, a partir de uma classe mais genérica (superclasse), herdando todas as características dessa classe.

No caso do exemplo anterior, poderíamos pensar que a classe Janela é uma especialização de uma classe mais genérica, como a classe Retângulo, que cria, desenha e move retângulos na tela. Como as janelas em geral são retângulos e as suas operações básicas são as mesmas, poderíamos criar a classe Janela a partir da classe Retângulo.

Com o mecanismo de herança, ao se criar uma classe a partir de outra classe, a nova classe (subclasse da classe original) herda todas as suas características. Isto significa que ao se definir uma subclasse, esta automaticamente ganha (herda) todos os atributos e operações da(s) classe(s) mãe(s) (superclasses). Neste caso, Janela é subclasse de Retângulo. Do mesmo modo Retângulo é superclasse de Janela.

Esse mecanismo de herança é conhecido como generalização/especialização. Isto é, a partir de uma classe mais genérica (generalização) como a classe Retângulo, que possui características e operações comuns ao desenho e movimentação de retângulos na tela, pode-se criar a subclasse Janela. Esta subclasse, além de herdar as características genéricas de Retângulo poderia ter outras características mais específicas (especialização) às janelas (Figura 4).

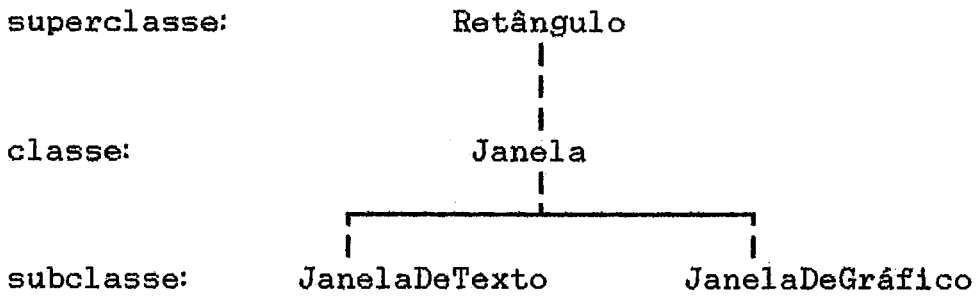

Figura 4 - Hierarquia de generalização/especialização

Herança Múltipla

Criação de uma subclasse, a partir de mais de uma classe diferente herdando as características de todas elas.

Quando uma classe tem mais de uma superclasse, é dito haver herança múltipla. Seguindo o mesmo exemplo, poderia se criar uma classe JanelaDeTexto que poderia ser subclasse de Janela e, ao mesmo tempo, por manipular cadeias de texto, também ser subclasse da classe CadeiaDeTexto (Figura 5).

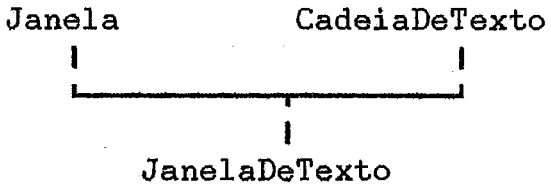

Figura 5 - Herança Múltipla

Acoplamento Dinâmico

Acoplamento dinâmico significa que o acoplamento de uma mensagem com seu método correspondente é feito em tempo de execução. Com isso, uma mesma mensagem pode ser tratada diferentemente de acordo com o objeto que for recebê-la. Um objeto pode enviar mensagens para objetos a serem criados posteriormente. Por exemplo, no caso de gerenciamento de janelas sobrepostas (Figura 6).

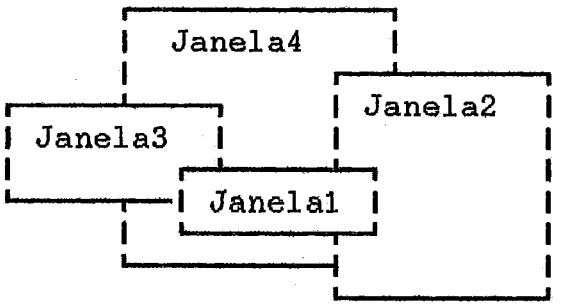

Figura 6 - Janelas Sobrepostas

Em sistemas cuja interface é baseada em janelas, o usuário pode abrir diversas janelas ao mesmo tempo e movê-las na tela, inclusive sobrepondo-as. Nesses sistemas é necessário que haja um mecanismo para gerenciar essas janelas.

Uma maneira é de se encadear as janelas que forem sendo abertas na tela, em uma lista encabeçada pela janela da frente. Frequentemente, porém, as janelas abertas possuem informações distintas, ou seja, cada janela pode ser de uma classe diferente: contendo apenas texto, contendo objetos gráficos,etc.

As **Janelas** abertas vão sendo organizadas em uma lista, como na figura 7.

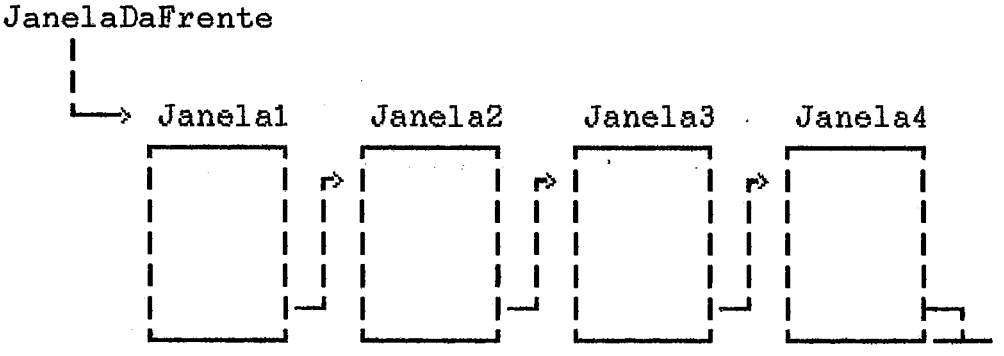

Figura 7 - Lista de Janelas

Caso a janela da frente seja fechada, as demais serão redescobertas e precisarão ser redesenhadas. Como as

janelas são de tipos diferentes, cada uma tem um modo de se redesenhar (texto, gráfico, etc.). Em uma linguagem tradicional, em que os tipos de todas as variáveis são definidos em tempo de compilação, teríamos que definir uma variável para cada tipo de janela (JanelaDeTexto, JanelaDeGráfico, etc.). Na orientação a objetos não há o conceito de tipos, i.e., uma mesma variável pode conter um inteiro, uma cadeia de texto (string), uma janela, etc., durante a execução de um programa. Deste modo, utilizando-se uma LPOO, pode-se ter a mesma variável apontando para todos os tipos de janela.

A seguir, apresentaremos dois algoritmos para resolver esse problema. O primeiro, no caso de se usar uma linguagem tradicional e o segundo usando-se uma LPOO. No primeiro caso, uma das maneiras mais otimizadas seia um único procedimento para redesenhar janela que, de acordo com o seu tipo, executaria o trecho correspondente.

Redesenha:

```
Case Janela.tipo faz
    texto: -----
    gráficos: -----
        .
        .
        .
    default: ------
Fim-case
```

No segundo caso, poderia-se utilizar uma única variável e em tempo de execução a mesma mensagem de Redesenhar, iria para o objeto que a variável estivesse apontando naquele momento.

```
J: = JanelaDaFrente;
enquanto J <> nulo faz
    Redesenha ( J) ;
    J : = Prox ( J) ;
fim-enquanto;
```

Como não há conceito de tipos, uma única variável (o ponteiro J) pode apontar para tipos diferentes como JanelaDeTexto, JanelaDeGráfico, etc.

Considerações

i) No caso de linguagens tradicionais, todos os procedimentos de todas as janelas (contidos no "case") ficam na memória. Nas LPOOs, só o método (procedimento) do objeto requisitado será carregado.

ii) Caso se deseje criar um novo tipo de janela, usando as linguagens tradicionais, deve-se criar mais uma opção dentro do "case", para tratar este novo tipo de janela, além de ter que se definir mais um tipo de variável para a nova janela e compilar tudo de novo. Já em uma LPOO, cria-se uma nova classe de janela, com seu método específico para redesenhar, sem alterar nenhuma outra parte e tudo continua a funcionar, pois a mesma variável poderá apontar para a nova janela e a mesma mensagem será enviada para esta janela também, que executará o seu método correspondente.

Ocultação da Informação

Permite que a estrutura interna (estrutura de dados, algoritmos, etc.) de um objeto seja inacessível (ocultada) só sendo alcançada através de suas operações.

II.2. Programação Orientada a Objetos

Em 1982, RENTSCH [15] fez uma previsão de que a Programação Orientada a Objetos (POO) seria nos anos 80 o que a programação estruturada foi nos anos 70. Realmente, de lá para cá, de acordo com STEFIK & BOBROW [16], o número de pesquisas e linguagens que surgiram introduzindo esses conceitos foi muito grande, comprovando o sucesso dessa técnica.

Ainda hoje, há uma grande polêmica sobre quais os conceitos fundamentais para o estilo de POO. Uma linguagem que suporta orientação a objetos, de acordo com PASCOE [17], tem que ter quatro elementos: ocultação de informação, abstração de dados, acoplamento dinâmico e herança. Já HALBERT & O'BRIEN [18], acreditam que essenciais são os conceitos de herança e tipos hierárquicos. Apesar das divergências, observamos que a linguagem SMALLTALK é considerada uma das mais importantes e que melhor tem atendido a esse estilo de programação [19].

Porém, independentemente dos conceitos a serem seguidos, o objetivo maior é obter as vantagens da Programação Orientada a Objetos tais como:

- **manutenibilidade** -> uma vez que há um baixo acoplamento entre os objetos, as alterações são feitas em um único lugar. Deste modo, fica mais fácil a manutenção de um sistema;

- **reusabilidade** -> com o encapsulamento, dados e todas as operações associadas a estes dados, ficam em um único objeto. Sendo assim, é mais fácil se reaproveitar software, pois no caso de se criar um sistema baseado em janelas, conforme os exemplos anteriores, pode-se reusar a classe janela. Nesta classe já estarão todas as operações básicas para se manipular uma janela;

- **extensibilidade** -> devido às características citadas nos dois itens anteriores, facilmente pode-se estender o sistema, criando novas classes sem que se tenha que alterar as outras partes do sistema que já estão funcionando.

Atualmente existem inúmeras linguagens e ambientes de programação que suportam, ou tentam suportar, os conceitos da orientação a objetos. Algumas dessas linguagens e ambientes são: Smalltalk, Actor, Eiffel, Objective-C, ++, ADA, Flavors, Loops, Object-Pascal, Neon, Modula-2, Mesa, Cedar, TRELLIS/OWL, ABCL, etc.

A seguir, descrevemos brevemente algumas dessas linguagens e ambientes de programação, dando uma ênfase maior ao ambiente de programação Smalltalk, por ser o pioneiro, e ao ambiente Actor, onde será implementado o ambiente TABA_OBJ.

II.2.1 - Algumas Linguagens Orientadas a Objetos

ADA

É uma das linguagens mais polêmicas da orientação a objetos, de acordo com TOUATI [20]. Foi baseada nela que

uma das propostas para métodos de desenvolvimento de software orientado a objetos mais citados, a proposta de BOOCH [3], foi feita.

Para TAKAHASHI [14], "Ada tem duas construções **package** e **task**, que garantem o encapsulamento de dados e que poderiam representar classes. Todavia, o mecanismo de **package**, embora permita a implementação de um tipo abstrato de dados, não caracteriza um tipo por si, enquanto o ideal seria que houvesse uma identificação plena entre o conceito sintático de **módulo** e o conceito semântico de **classe**. Por outro lado, o mecanismo de **task** reune diversas propriedades ideais de objetos: **tasks** podem ser criadas dinamicamente, passadas como parâmetros, e fazer parte de estruturas mais complexas envolvendo matrizes("arrays") e ponteiros ("pointers"). Há restrições sérias: não se pode fazer atribuição de **tasks**, nem testes essenciais a objetos tais como comparação, entre variáveis do tipo **task**. Não há suporte a herança em Ada. Finalmente, não há acoplamento dinâmico ".

Objective-C

Desenvolvido por COX, [21], [22], [23], [24] e [25], Objective-C é um precompilador de C que provê facilidades da orientação a objetos. Possui acoplamento dinâmico e herança, sendo possível implementar herança múltipla e "garbage collection". Em Objective-C as mensagens são escritas dentro de um programa comum em C, só que envolvidas por colchetes.

Junto com Objective-C, vem uma biblioteca de classes reusáveis, a "Software-ICs" (Circuitos Integrados de Software), que permite o aproveitamento de grande parte do código dos programas a serem gerados. Para COX [24], assim como os *chips* de silicone ajudam os engenheiros de harware a reusar o trabalho dos projetistas de *chips*, a programação orientada a objetos deve ajudar os programadores a reusar o código já existente.

C++

Desenvolvida nos Laboratórios Bells (AT&T) por STROUSTRUP [26] e [27], consiste de um preprocessador da linguagem C que permite a criação de classes através de um tipo novo **class** [28].

Através do tipo **class**, são definidas suas operações. É permitido fazer sobreposição de operadores, i.e., redefinir um operador já existente dentro de uma classe que funcione de maneira diferente (ex. na classe matriz, definir um operador "+" que faça soma de matrizes).

O compilador de C++ já foi desenvolvido por diversos fabricantes. A versão 2.0 da AT&T suporta herança múltpla e acoplamento dinâmico, porém todo o gerenciamento de memória e "garbage collection" tem que ser feito pelo programador [29]. Outro fabricante (Zortech Inc.) oferece além do compilador C++, uma biblioteca de classes que efetuam diversas tarefas (edição de textos, gerenciamento de janelas, busca binária, etc.).

Eiffel

A linguagem Eiffel faz parte de um ambiente desenvolvido por MEYER [30], [31], [32] e [33] no Interactive Software Engineering, que possui também um método para projeto, uma biblioteca e um conjunto de ferramentas. A biblioteca Eiffel possui um conjunto de classes que implementam as estruturas de dados mais importantes com suas respectivas operações, permitindo reusabilidade e extensibilidade.

Eiffel possui herança múltipla e generalização. É uma linguagem tipada em que toda a checagem de tipos pode ser feita estaticamente. Dentre as ferramentas do ambiente Eiffel, há um gerenciador de memória automático, depurador simbólico, "tracing" de execução, etc.

Trellis

O Ambiente de Programação Trellis [34], foi desenvolvido pela Digital Equipment Corporation. Trellis suporta a programação em **Trellis/Owl**, uma liguagem baseada em objetos, que permite herança múltipla e verificação de tipos em tempo de compilação (acoplamento estático). O ambiente é composto de uma série de ferramentas integradas que permitem editar, consultar, fazer compilações incrementais e depurar código. As ferramentas possuem uma interface consistente e compartilham uma base de dados comum ao ambiente.

A base de dados do **Trellis** armazena informações sobre o estado de compilação dos programas, tais como códigos fonte e objeto, referências cruzadas, etc. O compilador incremental atualiza a base de dados e com isso, **Trellis** pode oferecer ferramentas que armazenam e informam todos os erros e inconsistências dos programas. Porém, até o momento não há ainda uma biblioteca de classes pronta para ser reutilizada.

II.2.2 - A linguagem SMALLTALK-80 [1] e [19]

A programação em SMALLTALK-80 consiste em se identificar objetos, classificá-los de acordo com semelhanças e diferenças e projetar a linguagem de interação entre os objetos. A seguir abordamos rapidamente alguns de seus conceitos básicos.

Os principais elementos da linguagem são: objeto, classe, instância, mensagem e método. Seus conceitos são os mesmos das definições da seção II.1.1. Um **objeto** é dito ser uma **instância** de uma **classe** e um **método** descreve como o **objeto** deve responder à **mensagem**.

Uma classe é composta de: nome, superclasse, nomes de variáveis de instância e método.

O SMALLTALK-80 implementa o conceito de herança, isto é, pode-se criar uma subclasse a partir de uma classe, herdando as informações contidas na classe, podendo-se incluir novos métodos ou modificar métodos já existentes.

Ao receber uma mensagem, o objeto verifica se possui o método (a função) solicitado pela mensagem (de acordo com seu padrão) e, caso positivo, executa-a.

II.2.2.1 - O Ambiente SMALLTALK-80

O sistema SMALLTALK-80 consiste de uma linguagem de Programação Orientada a Objetos e de um conjunto de ferramentas integradas para interagir com os componentes da linguagem [35] e [36].

Todas as informações do sistema são representadas por objetos. A interface desse sistema é bastante amigável e toda baseada em janelas e cardápios, tendo o "mouse" grande utilização.

Ferramentas

O Sistema SMALLTALK-80 possui um conjunto de ferramentas que facilitam a tarefa de programação. Algumas delas são:

. **Projetos**

Permite que se gerencie várias tarefas de programação (em vários projetos) ao mesmo tempo, manipulando um **Projeto** de cada vez. Ao se mudar de um **Projeto** para outro, as informações aparecem como tinham sido deixadas na última vez em que esse projeto foi acessado.

Qualquer classe criada num **Projeto** é imediatamente disponível para todos os projetos do sistema. Dentro de um **Projeto** pode-se abrir janelas com **Áreas de Trabalho**, para auxiliar na manipulação dos dados, avaliação de expressões, etc. A **Área de Trabalho** provê várias máscaras para

facilitar a declaração das expressões criadas pelo usuário, além de máscaras para variáveis globais.

. **Browser**

Ferramenta poderosa que permite ter informações sobre a interface de um objeto. Dá acesso a todas as descrições de classes disponíveis no sistema, incluindo comentários sobre classes e métodos, além de exemplos de como usar várias classes.

É dividido em categorias que organizam as classes dentro do sistema e nas que organizam mensagens de cada classe. Com o **Browser** pode-se obter informações como:

- comentário sobre o papel de cada classe.

- descrição da parte da hierarquia de classes do sistema em que a classe se encontra,

- descrição das variáveis da classe,

- descrição das mensagens e métodos da classe (incluindo comentários),

- classificação da classe, em relação a outras classes,

- classificação das mensagens da classe,

- acesso a todos os métodos que enviaram uma mensagem,

- acesso a todas as mensagens enviadas por um método.

Além disso, o **Browser** também fornece uma máscara para auxiliar a definição de novas classes e mensagens.

Existem vários **Browsers** de classes de acordo com o subconjunto de classes acessíveis. O comando **Spawn** permite que se crie um **Browser** em que somente as informações da subparte do sistema selecionada é acessada (ex. Browser de Categorias de Classes do Sistema, Browser de Classes,

Browser de Categorias de Mensagens, Browser de Mensagens, Browser de Hierarquia de Classes).

. **Editor de Formas/Bits**

Provê suporte para criação de imagens gráficas tanto por desenho a mão livre, com uso do "mouse" (**Editor de Formas**), como através de métodos em que objetos gráficos recebem mensagens para construir uma imagem. Imagens simples podem ser representadas como instâncias de Forma, que têm altura e largura e um bit map indicando as regiões pretas e brancas (**Editor de bits**).

Imagens complexas podem ser representadas de dois modos: por uma forma muito grande ou por uma estrutura que inclua várias formas e regras para combinar e repeti-las de modo a produzir a imagem desejada.

O editor de Bits enfoca a criação/alteração de formas através de se especificar cada ponto (bit) como branco ou preto.

. **Depurador**

Pode apresentar algumas ou todas as sequências das mensagens enviadas antes da interrupção, conforme o selecionado. Permite selecionar cada uma para ver o método e em que ponto do método ocorreu a interrupção. Pode-se escolher qualquer mensagem na pilha e continuar a execução a partir desse ponto ou pular algumas mensagens, checar o valor das variáveis para tentar detectar o erro. Pode-se também alterar o valor das variáveis e prosseguir e ainda editar e recompilar um método dentro do **Depurador**.

. **Inspetor**

Permite obter-se informações sobre o estado interno de um objeto. Em uma janela mostra a lista das variáveis da instância a ser "inspecionada" e na outra exibe o valor da variável selecionada. Outra utilidade dessa ferramenta é a de testar o envio de uma mensagem para o objeto

inspecionado ou para testar a mensagem que avalia expressões.

Além dessas ferramentas, o Ambiente SMALLTALK-80 provê facilidades como um editor de texto poderoso e com interface amigável, comandos de **Comentários** (descreve o objetivo da classe ou do método) e **Explicações** (descrição sumária do papel de qualquer token que tenha sido selecionado).

Possui ainda um mecanismo para correção ortográfica e um dicionário compartilhado ("pooled dictionary") para as classes, contendo as variáveis que são acessíveis pelos métodos da classe e por suas subclasses.

As mensagens, além de serem enviadas para os objetos, podem, também, ser enviadas para classes. Nesse caso são mensagens para criação de instâncias, comentários sobre documentação, mensagens para criar variáveis de classe, criar constantes e mensagens para consultas.

II.2.3. A linguagem Actor

Actor é uma linguagem de programação orientada a objetos, cuja filosofia é bastante semelhante ao **Smalltalk**. A programação é toda feita em termos de objetos e das mensagens que estes objetos enviam e recebem.

Tudo em **Actor** é um objeto [11]. Números, caracteres, vetores (arrays), cadeias de caracteres, aplicações, janelas, métodos, etc., são todos objetos.

Toda ação que ocorre em **Actor** (com exceção das chamadas ao MS-Windows ou MS-DOS) é o resultado do envio de uma mensagem a um objeto, que responde executando o método solicitado.

Por seguir essas regras de modo rígido, **Actor** é dito ser uma linguagem puramente orientada a objetos, diferindo das linguagens híbridas que não seguem essa regra.

Em **Actor**, o acoplamento é dinâmico, o mecanismo de herança é de herança simples e há um esquema para "garbage collection".

II.2.3.1 O Ambiente Actor

Actor, além de uma linguagem de programação orientada a objetos, possui algumas ferramentas, constituindo, assim, um ambiente de programação. O Ambiente **Actor** usa toda a estrutura do produto Microsoft Windows (MS-Windows).

Inspetor

O **Inspetor** permite que um objeto seja examinado em detalhes, podendo-se inclusive enviar mensagens para o objeto inspecionado. A janela do **Inspetor** possui três janelas menores. A primeira janela contém os nomes das variáveis de instância do objeto inspecionado. A segunda contém índices ou chaves, no caso da classe inspecionada ser uma coleção. Ao selecionar-se qualquer desses itens, a terceira janela exibe o seu valor correspondente. Esta última janela é a janela de edição do **Inspetor**.

Com o **Inspetor** pode-se consultar os valores de quaisquer variáveis ou elementos de um objeto, assim como identificar sua classe. Pode-se inspecionar qualquer tipo de objeto, inclusive classes.

Browser

O **Browser** é uma ferramenta muito útil. Com ela pode-se ver o código fonte dos métodos que já vem com o sistema, pode-se saber inúmeras informações sobre qualquer classe do sistema, tais como quais variáveis de instância os objetos de uma determinada classe terão.

Através do **Browser** é possível:

- criar novos métodos em uma classe;
- editar métodos;

- criar novas classes,
- remover métodos ou classes do sistema,
- ver os métodos de classe e de objeto de uma determinada classe, etc.

Depurador

Permite que sejam depurados erros. A caixa de diálogo do Depurador apresenta uma lista de atividades que levaram àquele erro. A Janela do Depurador é dividida em quatro partes. A primeira parte contém as mensagens mais recentes que foram enviadas. A segunda mostra quem recebeu a mensagem e os parâmetros (caso existam) da mensagem. Se algum parâmetro for selecionado, a terceira janela exibirá seu valor. A quarta janela é a área de edição que mostra o código do método executado. Este código pode ser alterado e compilado no próprio Depurador.

Além dessas ferramentas, o Actor possui um gerador de analisadores sintáticos chamado YACC ("Yet Another Compiler Compiler"). Com o YACC, o usuário pode facilmente criar pequenas linguagens para serem usadas em suas aplicações. Há também uma linguagem para representação de frames a FRL ("Frame Representation Language"), que permite representar o conhecimento, armazenando objetos e os relacionamentos entre esses objetos.

II.2.4.Visão Geral

Ao apresentar a sua linguagem, Objective-C, COX [25] fez um quadro comparativo de quatro linguagens orientadas a objetos, com algumas das principais características da orientação a objetos. Foi feita uma tabela (Tabela 1) baseada neste quadro, comparando características de todas as linguagens apresentadas, visando concluir esse estudo.

	A.	S. O.	G. C.	H.	H. M.	D. C.	B. C.
ST-80	dinâm.	sim	sim	sim	sim	sim	sim
Obj-C	ambos	não	poss.	sim	poss.	sim	sim
ADA	estát.	sim	não	não	não	sim	não
Actor	dinâm.	sim	sim	sim	não	sim	sim
Eiffel	estát.	sim	sim	sim	sim	sim	sim
C++	ambos	sim	não	sim	não	sim	sim*
Trellis	estát.	sim	sim	sim	sim	sim	não

Tabela 1 - Quadro Comparativo de Linguagens de Programação com Orientação a Objeto

Legenda:

A. - Acoplamento

S. O. - Sobreposição de Operadores

G. C. - "Garbage Collection"

H. - Herança

H. M. - Herança Múltipla

D. C. - Disponibilidade Comercial

B. C. - Biblioteca de Classes

* A biblioteca de classes foi incluida na versão da Zortech Inc.

II.3 - Métodos de Desenvolvimento de Software [3], [4], [5], [6], [7], [37], [38] ,[39], [40], [41], [42], e [43]

Com o sucesso da Programação Orientada a Objetos (POO), tornou-se necessária uma nova maneira de se desenvolver software e passou-se a aplicar esses conceitos em métodos de desenvolvimento de software.

Algumas das características do desenvolvimento orientado a objetos são:

- visibilidade: gerar software de acordo com os requisitos de modo mais simples, trabalhando com objetos e mensagens em todos os níveis de abstração, desde a especificação até a implementação.

- alterabilidade: como a visibilidade é maior, fica mais fácil olhar a Especificação de Requisitos e imaginar o que deverá ser feito para se alterar uma parte do sistema. Por outro lado, quando é feita uma alteração na implementação também é fácil imaginar o seu efeito na especificação do sistema;

- extensibilidade: é muito mais fácil estender um sistema orientado a objetos, já que eles são bastante modulares. De acordo com os critérios de YOURDON [44], o mais alto nível de modularidade corresponde a alta coesão e baixo acoplamento entre os módulos. Isto é fácil na orientação a objetos uma vez que os objetos são naturalmente coesos já que encapsulam dados e procedimentos. Não obstante, pelo fato dos objetos só terem acesso aos outros objetos através da interface, há um baixo acoplamento entre eles;

- reusabilidade: Devido à modularidade dos sistemas orientados a objetos, fica mais fácil aproveitar partes de sistemas já construidos, sem que se tenha muito trabalho. A tendência, com o tempo, é cada vez mais, desenvolver menos e aproveitar mais o código já existente.

Seguindo essa filosofia, foram feitas inúmeras propostas para desenvolvimento de software com orientação a objetos. Porém, a maioria destas propostas foi feita para apenas uma das fases de desenvolvimento. Existem algumas propostas para a fase de projeto, outras para a fase de especificação e pouquíssimas que cobrem todo o desenvolvimento. Essas propostas ainda não estão consolidadas e a maioria sugere o uso da intuição em determinadas etapas. A seguir serão apresentadas algumas dessas propostas para desenvolvimento de software orientado a objetos encontradas na literatura.

II.3.1 Proposta de Booch

Foi proposto por BOOCH [37] e [3], um método de desenvolvimento de software que consiste de uma seqüência de passos a serem seguidos e repetidos, sucessivamente, até que se tenha alcançado o nível de projeto desejado. Os passos são os seguintes:

1 - Definir o problema.

2 - Desenvolver uma estratégia informal para a solução do problema.

3 - Formalizar essa estratégia repetindo os seguintes passos até a modularização completa:

a) Identificar as Classes e Objetos da solução, recolhendo-se os substantivos existentes na descrição

informal, levando-se em consideração a ocorrência de substantivos sinônimos que identifiquem o mesmo Objeto/Classe.

b) **Identificar os atributos dos Objetos**, recolhendo-se os adjetivos associados aos Objetos identificados.

c) **Identificar as operacões nos Objetos**, selecionando os verbos - ações - ligados aos Objetos que as sofrem ou realizam.

d) **Estabelecer a visão externa das Classes**, formalizando o conjunto de operacões e a tipificação dos operandos de entrada e saída. Além disso, deve-se identificar a visibilidade das classes, ou seja, que classes vêem que classes, permitindo-se traçar a topologia das classes do sistema.

e) **Estabelecer a visão interna das Classes**, definindo a estrutura de dados de cada objeto da classe e a implementação de suas operações.

Para o passo 1 (definição do problema), que possui um nível maior de abstração, Booch sugere que se utilizem instrumentos com que se tenham maior familiaridade, tais como os Diagramas de Fluxo de Dados (DFDs) da Análise Estruturada de GANE [45]. Desse modo, se torna mais fácil a visualizacão e definição do problema.

Já a estratégia informal seria feita através de um texto de onde facilmente seriam identificados os objetos (os substantivos desse texto) e suas operações (os verbos).

A seguir, vão se construindo os objetos, através da definição dos algoritmos de suas operacões e da especificação de suas interfaces. Segue-se esse processo recursivamente até que se tenha atingido o nível de detalhamento desejado. A partir daí, a implementação na linguagem ADA é quase direta.

Booch propôs que se usassem diagramas para simbolizar os objetos e sua troca de mensagens, baseado na notação usada para representar os "packages" da linguagem ADA. Esses diagramas são conhecidos como os Diagramas de Booch (Figura 8).

Neste diagrama, os objetos são simbolizados por retângulos contendo pequenos retângulos no lado esquerdo que servem para representar a interface deste objeto. Os retângulos arredondados contêm os seus atributos (as variáveis de instância) e os outros retângulos contêm suas operações. As setas ligando um objeto a outro representam as mensagens que um objeto envia para outro solicitando que este efetue uma de suas operações.

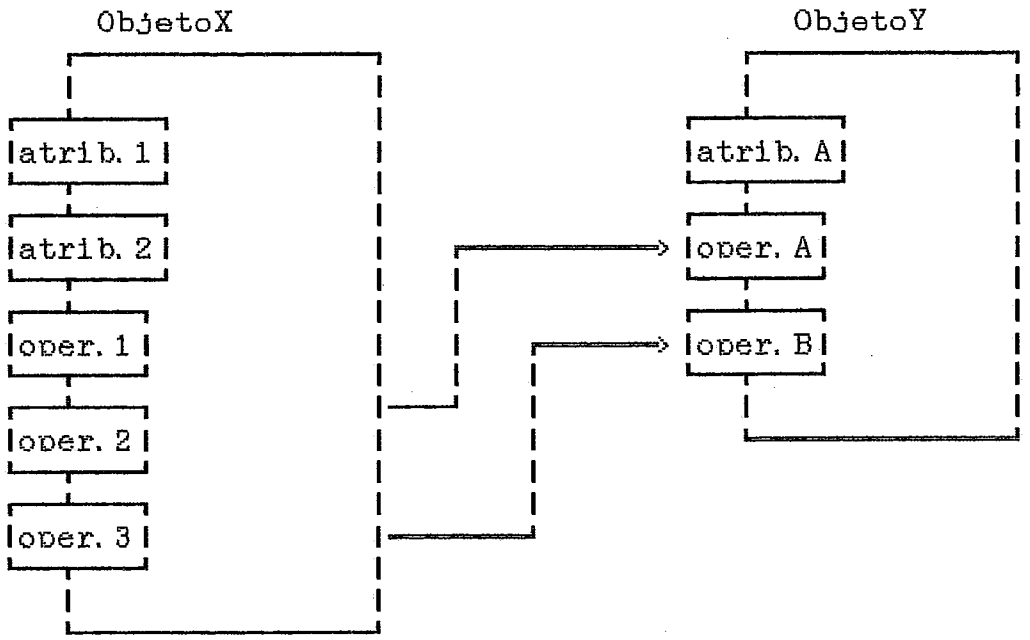

Figura 8 - Diagramas de Booch

A representação de troca de mensagens tende a ficar bastante confusa, uma vez que cada operação de um objeto, costuma enviar diversas mensagens para outros objetos. Todavia, este diagrama é uma das pouquíssimas propostas de diagramas para dar suporte ao desenvolvimento com orientação a objetos e é bastante usado.

II.3.2 Proposta de Lorensen [4]

Um outro método mais voltado para linguagens realmente orientadas a objetos (i.e. que suportam os conceitos de herança, mensagens, encapsulamento, etc.), apresentado por LORENSEN e descrito por PRESSMAN [46], cobre somente a fase de projeto. O método parte de um documento (especificação de requisitos) já pronto e consiste de um processo recursivo que é basicamente o seguinte:

1 - Identificação das abstrações de dados para cada subsistema (que seriam as classes do sistema).

Essa identificação deve ser feita de preferência de maneira "top-down", apesar de algumas vezes as abstrações já estarem mencionadas explicitamente na especificação de requisitos [46]. É considerado o passo mais difícil desse método para projeto e a seleção dessas abstrações influencia toda a arquitetura do sistema.

2 - Identificação dos atributos de cada classe (as variá variáveis de instância).

No caso das classes corresponderem a objetos físicos, os seus atributos são facilmente identificáveis. Porém, outras variáveis de instância poderão ser necessárias para atender aos requisitos de outros objetos do sistema. Portanto, deve-se adiar a especificação das estruturas de dados contendo os atributos até a fase de projeto detalhado.

3 - Identificação das operações de cada classe.

As operações seriam os métodos das classes. Não se deve especificar os detalhes da implementação dos métodos, somente sua funcionalidade. Caso a nova classe seja subclasse de outra já existente, verificar os métodos da classe mãe para ver se é necessário reescrever algum deles para se adequar às particularidades da nova subclasse. Adiar o projeto interno dos métodos para a fase de projeto

detalhado, onde técnicas mais convencionais de projeto podem ser usadas.

4 - Identificacão da comunicacão entre os objetos, através de mensagens, e teste dos objetos através de cenários.

Cenários consistem de mensagens para objetos, verificando se o projeto está de acordo com a especificação. Ou seja, um grupo de mensagens é enviada para cada objeto, para verificar se o objeto está executando seus métodos corretamente e se estes correspondem ao solicitado pelo usuário.

5 - Aplicar heranca, onde for adequado.

Se o processo de abstração do passo 1 for feito de modo top-down, então a herança deve ser introduzida lá. Se as classes tiverem sido criadas de modo "bottom-up" (caso os requisitos já as tenham definido diretamente) então a herança deve ser aplicada nesse passo, antes de se entrar em um nível de detalhamento maior. O objetivo é de se reusar o máximo possível as classes e métodos já projetados.

Lorensen propõe que se decomponha o sistema em unidades menores (subsistemas) e mais fáceis de se visualizar e identificar seus objetos ou classes. A partir daí, vai-se identificando atributos, interfaces e operações, que por sua vez podem gerar novos objetos e assim por diante para cada subsistema.

Esse processo recursivo se repete até que se tenha atingido o nível de detalhamento desejado. Pelo fato do método cobrir somente a fase de projeto é necessário também que ele seja usado em conjunto com um método para a análise e especificação de requisitos. Porém, nenhum método em particular é sugerido pelo autor.

II.3.3 Método para Especificação de Bailin

Este método para a fase de especificação tem o objetivo de evitar o que muitos métodos para a fase de projeto sugerem, isto é, que se faça a especificação usando métodos tradicionais como a Análise Estruturada [45] e a partir daí fazer o projeto orientado a objetos. BAILIN [5] afirma que o princípio de agregação da análise estruturada é diferente do da orientação a objetos, já que a análise estruturada agrupa as funções que são componentes de uma função de mais alto nível. Por outro lado, a orientação a objetos agrupa as funções que operam em uma mesma abstração de dados.

A idéia do método é aproveitar o conhecimento que as pessoas têm da Análise Estruturada para melhor identificar os requisitos e a partir daí, com o auxílio do diagrama de entidades-relacionamento, desenhar o EDFD (Diagrama de Fluxo de Dados e Entidades, proposto por SEIDEWITZ & STARK [47]). Através desse diagrama, com auxílio de uma ferramenta automatizada geradora de projeto, a passagem para a fase de projeto é imediata, sendo gerados *objetos ordinários* em um diagrama de objetos, que correspondem aos "packages" da linguagem ADA. Os principais passos desse método são:

- identificar as entidades chaves do domínio do problema;
- distinguir entre entidades ativas e passivas;
- estabelecer fluxo de dados entre as entidades ativas;
- decompor as entidades (ou funções) em sub-entidades e/ou funções;
- verificar se existem novas entidades;
- agrupar funções em novas entidades;
- atribuir novas entidades a domínios adequados.

II.3.4 Proposta de Rotemberg

Um outro método proposto por ROTEMBERG [7], [38] e [39], em sua tese de mestrado, cobre todo o ciclo de vida. Para a fase de especificação são oferecidas duas alternativas.

A primeira alternativa utiliza, diretamente, uma linguagem que expressa o modelo de objetos. Para essa alternativa, tenta-se definir o problema e a partir daí fazer uma lista inicial dos requisitos desejados. A seguir:

a) identificam-se as entidades e ações, selecionando somente as que forem relevantes ao problema;

b) identificam-se os objetos, associando as ações às entidades que as sofrem ou realizam;

c) identificam-se as entidades externas;

d) representam-se os objetos, suas operações (ações) e as entidades externas;

e) completa-se a representação com os relacionamentos e fluxos de dados;

f) compara-se o modelo com os requisitos do sistema;

g) definem-se as interfaces dos objetos.

A segunda alternativa utiliza diagramas de fluxo de dados que serão transformados, posteriormente, para uma linguagem que expresse o modelo de objetos. As etapas desta transformação são [7]:

a) identificam-se os objetos (os depósitos de dados são candidatos a dar origem aos objetos);

b) identificam-se as operações (ações) de cada objeto analisando-se os processos tentando associar as operações aos objetos;

c) representam-se os objetos, suas operações e as entidades externas;

d) completa-se a representação com os relacionamentos e fluxos de dados. Para isto, analisam-se os diagramas de fluxo de dados e o modelo de objetos que foi obtido, de maneira a identificar os fluxos de dados entre objetos e entre objetos e entidades externas. Os relacionamentos entre os processos serão indicados pelos fluxos de dados entre as operações dos objetos. O mesmo acontece entre processo e entidades externas, que terão seus relacionamentos indicados por fluxos de dados entre operações dos objetos e entidades externas.

e) definem-se as interfaces dos objetos. Operações que não recebem fluxo de dados ou só os recebem do objeto a que pertencem são candidatas a serem transformadas em operações privativas ao objeto.

Na fase de projeto, analisa-se o modelo em relação a sua hierarquia verificando a necessidade de generalização ou especialização de alguma classe. Além disso, **consulta-se a biblioteca de classes para uma possível reutilização.** Finalmente **identificam-se as trocas de mensagens entre objetos** e entre objetos e entidades externas.

No projeto detalhado, são definidas as classes (de acordo com a linguagem de implementação), projetam-se seus procedimentos (métodos) classificando-os em categorias (de classe, de instância, etc.).

Para Rotemberg, a vantagem desta segunda alternativa é que a maioria dos analistas já tem uma certa familiaridade com a Análise Estruturada e, portanto, a transição de uma abordagem para outra seria mais gradual. Todavia justamente pelo fato das duas abordagens modelarem o sistema de modo diferente (a primeira é funcional e a segunda orientada a objetos) a passagem da fase de análise para a fase de projeto se torna muito complexa.

II.3.5 - A proposta de Coad e Yourdon (OOA) [6]

A proposta de Análise Orientada a Objetos, OOA, feita por COAD & YOURDON [6] consiste basicamente de cinco passos:

- Identificar **Objetos**;
- Identificar **Estruturas**;
- Definir **Assuntos**;
- Definir **Atributos**;
- Definir **Serviços**.

Uma vez construído o modelo, ele é apresentado e revisto através de cinco camadas:

Camada de Assunto -> Assunto é um mecanismo para controlar quanto do modelo o leitor consegue visualizar de uma única vez. A seguir veremos um exemplo de uma representação gráfica da camada de assunto (Figura 9).

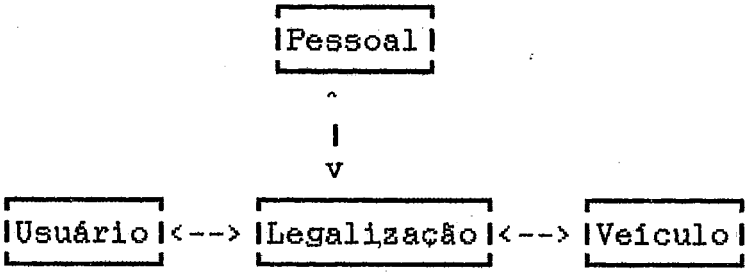

Figura 9 - Camada de Assunto

Camada de Objeto -> Nesta camada são identificados os objetos a partir dos assuntos da camada anterior. São postas Linhas de Assuntos, que são linhas que servem para delimitar os assuntos, para facilitar o leitor na navegação pelas diversas camadas.

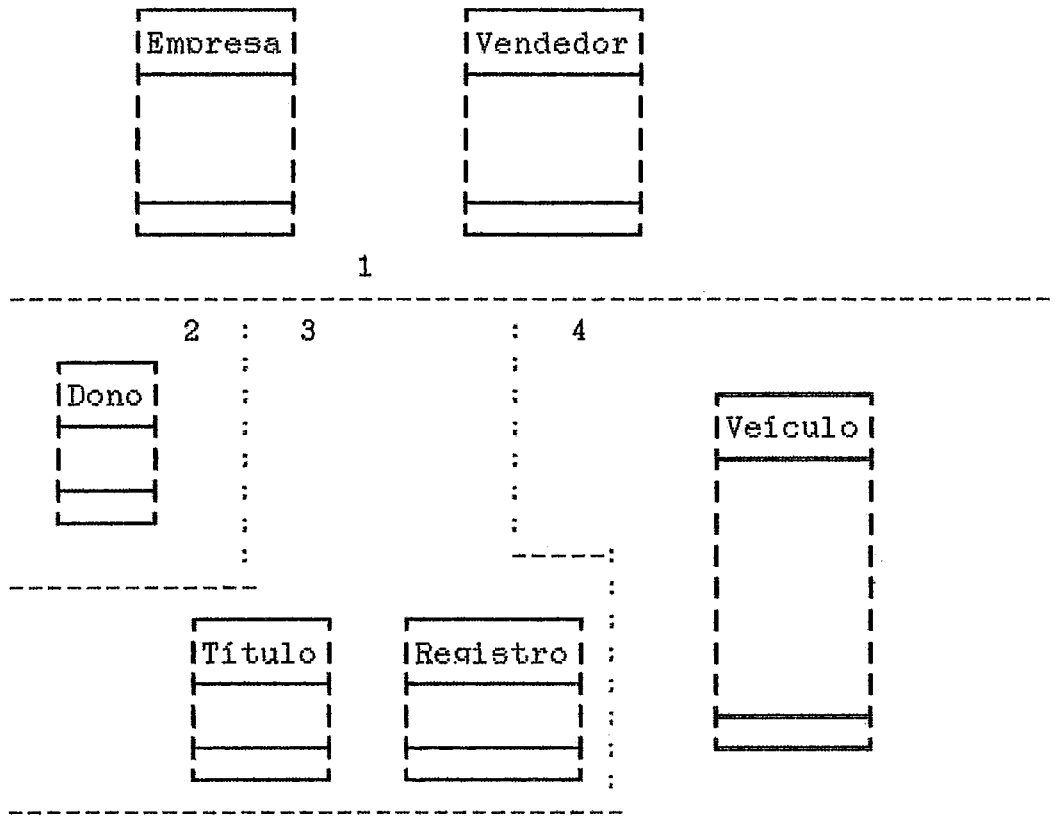

Figura 10 - Camada de objeto

Um objeto é representado através de um retângulo com bordas arredondadas e é dividido em três partes. A primeira parte contém o nome do objeto, a parte do meio contém seus atributos e a última parte contém seus serviços, que correspondem aos métodos (Figura 10). As duas últimas partes são preenchidas, respectivamente, nas camadas de atributos e serviços.

Camada de Estrutura -> Representa a complexidade do espaço do problema e é feita em duas etapas. A primeira etapa monta a **Estrutura de Classificação**, onde cada objeto é considerado uma generalização de outros objetos e depois uma especialização.

A segunda etapa constrói a **Estrutura de Montagem**, onde cada objeto é considerado como um todo e depois como uma parte componente. As duas estruturas são vistas na Figura 11.

Camada de Atributos -> Os atributos são listados na parte central dos símbolos de Objeto e especificados no **Repositório de Objetos**. Este repositório consiste das cinco camadas do OOA e de uma documentação de suporte baseada em gabaritos ("templates"). Os atributos comuns a todas as subclasses de uma determinada classe são listados na classe e automaticamente estendidos para as suas especializações na estrutura, isto é, nas subclasses. Caso algum destes atributos não se aplique a uma subclasse (especialização), este

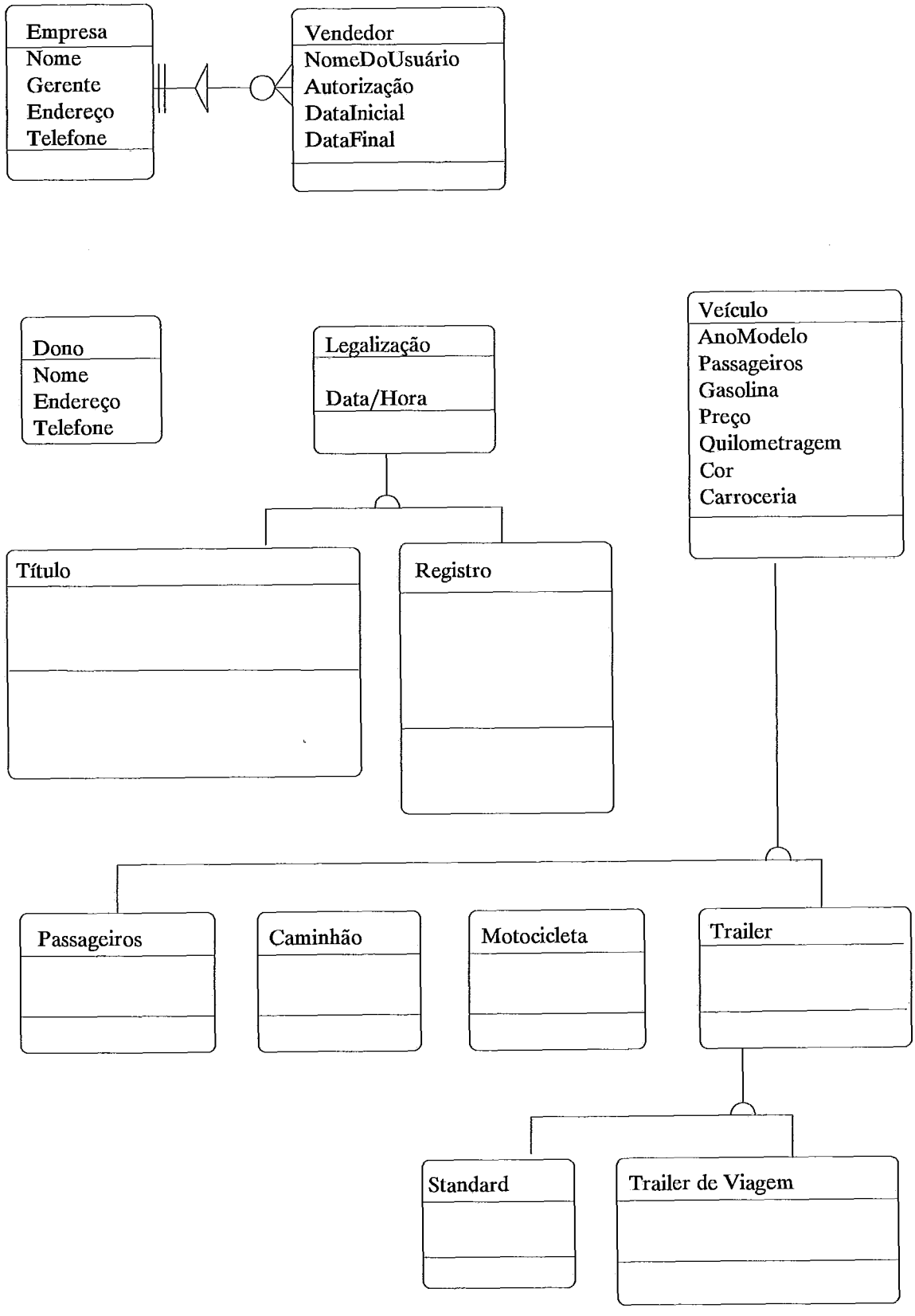

Figura 11 - Camada de Estrutura

atributo aparecerá com uma marca indicando sua ausência no símbolo desta subclasse.

Outro símbolo é usado para representar o mapeamento de uma instância de uma classe com outras instâncias. Este símbolo é denominado **Conexões de Instâncias** e indica multiplicidade e restrições de participação entre instâncias (Figura 12). O símbolo da esquerda, (O ou |), representa as restrições de participação (conexão opcional ou obrigatória) e o símbolo da direita (| ou <), representa a multiplicidade (existência de uma ou várias instâncias).

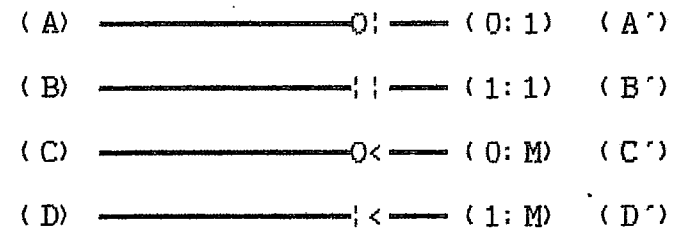

Figura 12 - Conexões de Instância

Na Figura 12, supondo que estas conxões estejam ligando dois objetos, a primeira conexão indica que para cada instância de A, tem uma instância de A' associada (multiplicidade), todavia, uma instância de A pode existir, sem que haja uma instância de A' (conexão opcional).

Já a última conexão, indica que para cada instância de D, tem M instâncias de D' associadas (multiplicidade) e que para haver uma instância de D, tem que haver uma instância de D' (conexão obrigatória).

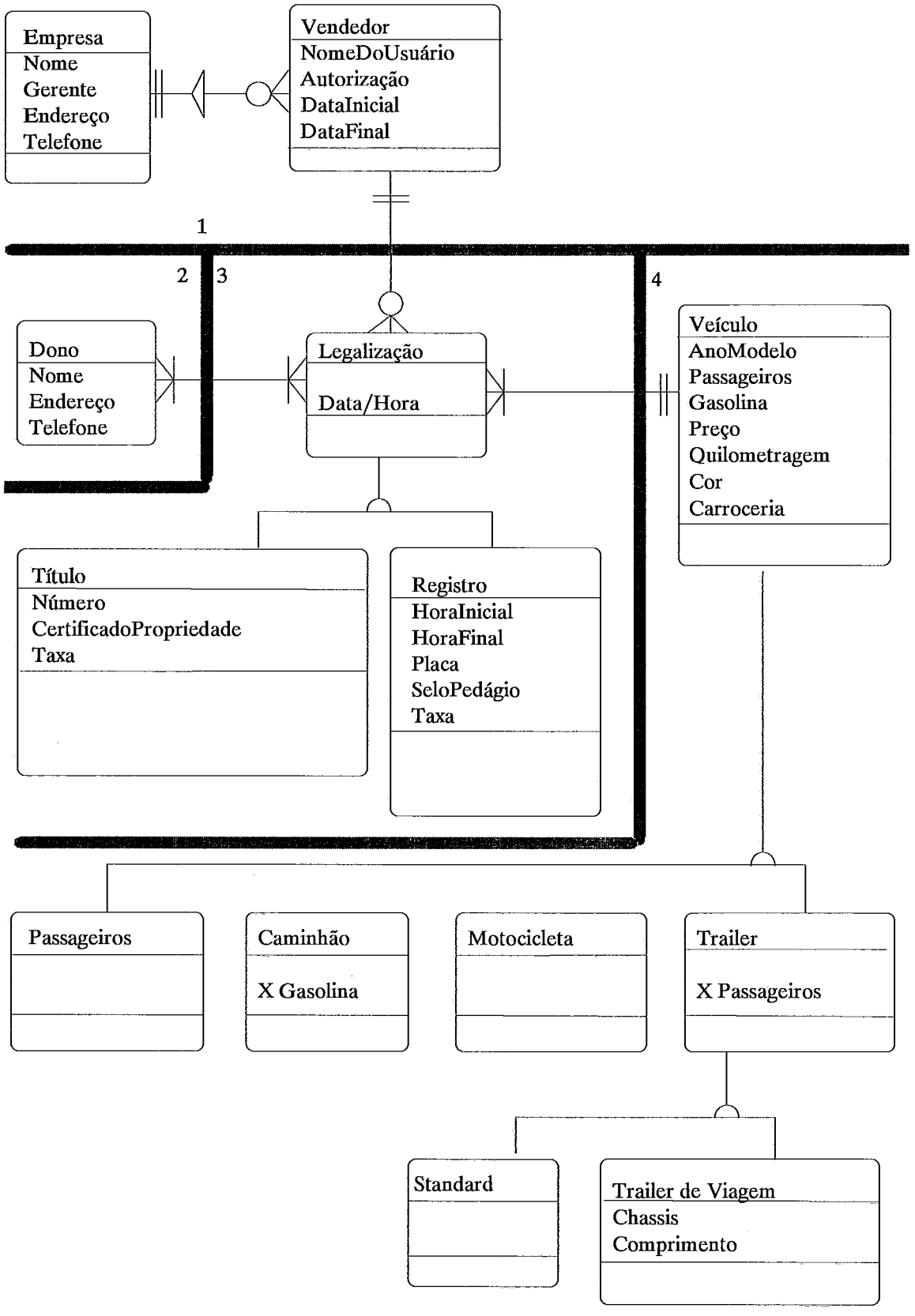

Figura 13 - Camada de Atributos

Um exemplo da Camada de Atributos é visto na Figura 13. O mapeamento existente entre o objeto **Empresa** e o objeto Vendedor, significa que, quanto às restrições de participação, para uma empresa existir, não é necessário que existam funcionários, porém, para haver um funcionário é necessário que haja uma empresa. Em termos de multiplicidade, o mapeamento indica que para cada instância de **Empresa**, pode haver várias instâncias de **Vendedor**, entretanto cada instância de Vendedor está associada somente a uma instância de **Empresa**.

Camada de Serviço -> Serviço é o processamento a ser efetuado quando chega uma mensagem (equivalente ao método). Serviços são identificados no diagrama e especificado no Repositório de Objetos. Serviços são listados na parte inferior dos símbolos de Objeto (Figura 14). Serviços para incluir, remover, alterar e selecionar instâncias são considerados serviços implícitos para todos objetos e, sendo assim, não aparecem no diagrama, mas são especificados no Repositório de Objetos. Serviços pertenentes a todas subclasses de uma classe são definidos na classe.

Conexões de Mensagens representam o envio de uma mensagem de um objeto para outro e são representados no diagrama por setas pontilhadas (Figura 14).

O diagrama proposto por COAD e YOURDON [6] é bastante rico, pois consegue representar muitas informações. Uma das propostas para diagramas de objetos, o Diagrama de Booch, só representa objetos com seus atributos e métodos e a troca de mensagens entre objetos. Este diagrama representa,

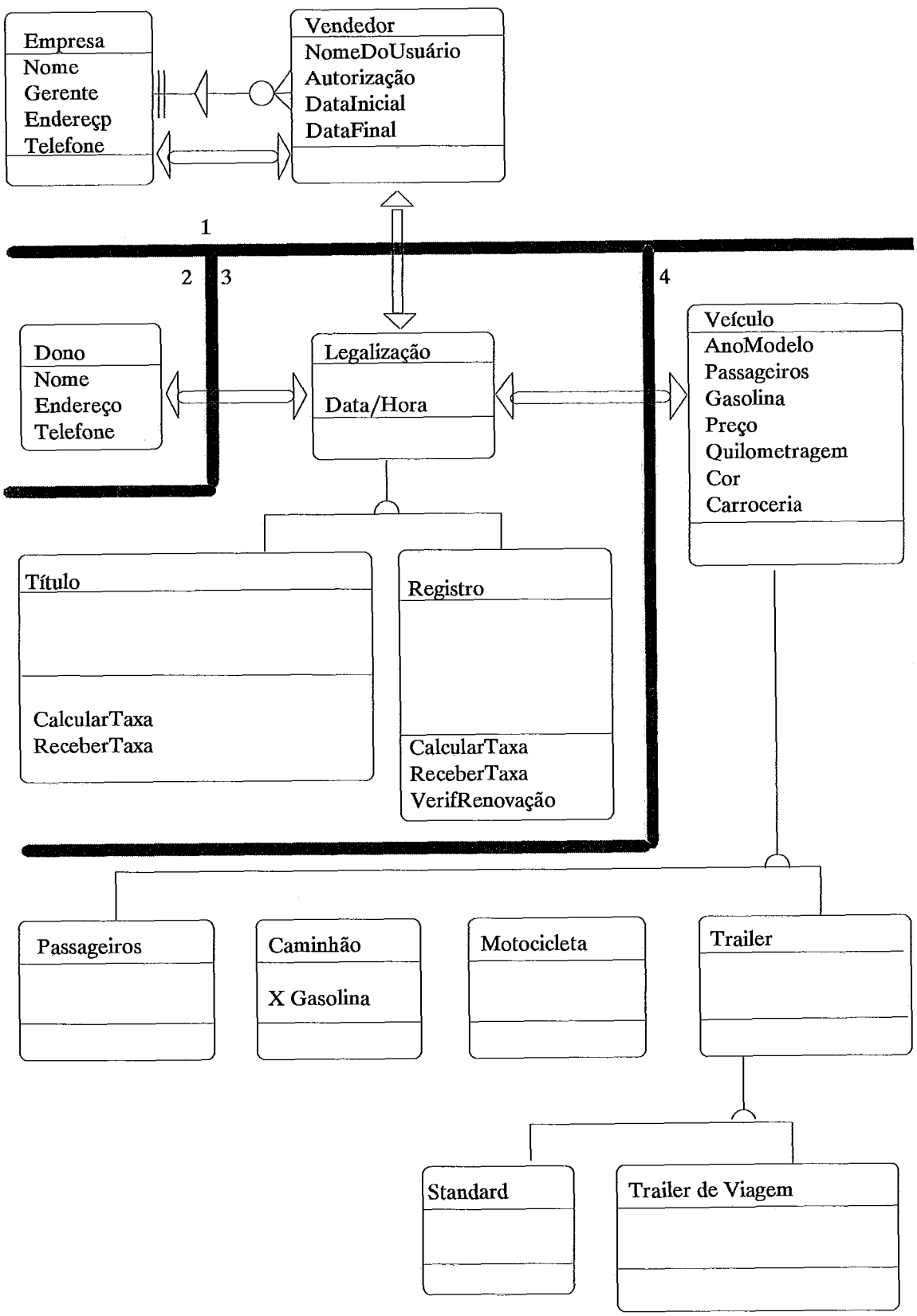

Figura 14 - Camada de Serviços

diagrama representa, além disso, a estrutura de classificação, organizando as classes e suas subclasses, a estrutura de montagem (objeto e seus componentes) e mapeamentos entre os objetos. Por outro lado, é uma das poucas propostas para diagrama de objetos bem definida existente.

II.3.6 - Considerações

Neste capítulo foram apresentados os conceitos básicos da orientação a objetos. Em seguida foram descritos algumas linguagens de programação com orientação a objetos e métodos de desenvolvimento de software com orientação a objetos.

Os métodos de desenvolvimento de software orientados a objetos são ditos serem muito dependentes da linguagem de programação a ser usada. Isto porque de acordo com a linguagem escolhida, se terão procedimentos diferentes na fase de projeto. No caso de se usar uma linguagem realmente orientada a objetos, que possua uma biblioteca de classes, não será necessário definir determinadas classes na fase de projeto. Neste caso, diversas classes poderão ser reutilizadas ou bastará definir-se algumas operações ou variáveis a partir de uma classe já existente. Caso contrário, se for usada uma linguagem que não possua uma biblioteca, todas as classes terão que ser definidas. Além disso, como a linguagem ADA não possui herança, nem troca de mensagens, etc., alguns artifícios têm que ser feitos na fase de projeto para simular esses conceitos, ou então, projetar-se sem usar esses conceitos. O mesmo cuidado deve-se ter com todas linguagens (Smalltalk, Objective-C, C++, etc.).

De todos os métodos apresentados não há algum que já seja bem consolidado. Todos são propostas intuitivas, buscando desenvolver software desde o início, de modo mais próximo de como enxergamos o mundo real. Todavia, o uso extensivo destas propostas poderá apontar suas deficiências, permitindo que elas sejam corrigidas.

A tendência com o tempo é ir-se aperfeiçoando cada vez mais as linguagens e métodos de modo a diminuir o "gap semântico". Deste modo, software será desenvolvido de maneira mais natural e atingirá, mais facilmente, os requisitos estabelecidos pelo usuário.

CAPÍTULO III - TABA_OBJ: UM AMBIENTE DE DESENVOLVIMENTO DE SOFTWARE COM ORIENTAÇÃO A OBJETOS

Este capítulo apresenta TABA_OBJ, um ambiente de desenvolvimento de software orientado a objetos para a estação TABA. Parte-se de uma justificativa de porque definiu-se um ambiente novo, ao invés de usar um ambiente já existente.

Em seguida é apresentada a definição de termos adotada no Projeto TABA e, consequentemente, neste trabalho, para ambientes de desenvolvimento de software. Para a definição do ambiente TABA_OBJ, são realizados diversos estudos de modelos de ciclo de vida, métodos e roteiros de documentação para ambientes encontrados na literatura. Finalmente, é definido o ambiente TABA_OBJ, envolvendo uma proposta de modelo de ciclo de vida a ser adotado, um método, instrumentos e roteiros de documentação.

III.1 - Justificativa da definição do Ambiente TABA_OBJ

O estudo realizado na literatura atual sobre orientação a objetos mostra-nos, em primeiro lugar, que ainda não foi proposto um ambiente de desenvolvimento de software completo, segundo o paradigma da orientação a objetos. Os sistemas que estão mais próximos disto são os ambientes de programação, que possuem editores, depuradores, dicionário de classes, enfim, um conjunto de ferramentas de apoio à fase de implementação de software [11], [32], [33] e [35]. O ambiente **Eiffel** [32] e [33] ainda propõe um método para projeto com orientação a objetos, além do ambiente de programação usado na fase de implementação.

Os métodos propostos, ainda são muito informais e a maioria cobre somente a fase de Projeto ou então de Análise. Não há muitas propostas de métodos que vão desde a

fase de Análise até a Implementação. A única encontrada foi a proposta de ROTEMBERG [7], [38] e [39].

Esta constatação, deu-nos uma grande motivação no sentido de estudar e propor um ambiente de desenvolvimento de software com orientação a objetos. A idéia inicial surgiu a partir de um trabalho com BLUM [9], onde foi elaborada uma proposta de desenvolvimento de software orientado a objetos. A partir daí, sentiu-se a necessidade de aperfeiçoar esta proposta e estendê-la para definição de um ambiente de desenvolvimento de software orientado a objetos que faria parte da estação TABA. Para permitir uma crítica da proposta, esta foi aplicada em dois projetos acadêmicos do curso Análise e Projeto de Sistemas I e II, durante o segundo semestre de 1988. A partir desta experiência, o método foi avaliado e alterou-se aspectos não satisfatórios [10]. Para completar a proposta do ambiente, fez-se um estudo das diferentes abordagens para modelos de ciclo de vida, métodos e roteiros de documentação. Finalmente, definiu-se o ambiente de desenvolvimento de software orientado a objetos TABA_OBJ e fez-se uma proposta de ferramentas do ambiente a serem implementadas.

III.2 - Ambientes de Desenvolvimento de Software

Na Engenharia de Software, existem diversos termos para os quais não existe consenso. Ambiente de Desenvolvimento de Software é um desses termos. Assim sendo, antes de apresentarmos o Ambiente TABA_OBJ, definimos, nesta seção, Ambiente de Desenvolvimento de Software segundo o enfoque adotado no Projeto TABA [48] e [49].

Um Ambiente de Desenvolvimento de Software é um conjunto integrado de métodos e ferramentas que suportam o desenvolvimento de um software nas diferentes etapas de seu ciclo de vida. Assim sendo, um Ambiente de Desenvolvimento

de Software é composto de:

- um **ciclo de vida**, que define as fases do processo de desenvolvimento e as atividades a serem realizadas em cada uma dessas fases;

- um conjunto de **métodos**, contendo diretivas para a seleção e aplicação sistemática de técnicas e instrumentos, de forma a organizar o pensamento e o trabalho do usuário ao longo do processo de desenvolvimento de software;

- um conjunto de **ferramentas** que automatizam os instrumentos do método.

O termo **ciclo de vida** é normalmente usado para definr as fases do processo de desenvolvimento de software, especificando todas as atividades que devem ser desempenhadas nesse processo. Em abordagens mais abrangentes, modelo de ciclo de vida têm sido usado não só para definir atividades a serem desempenhadas, mas também para orientar a gerência do próprio processo de desenvolvimento, abordando aspectos de custo, recursos alocados, tempo dispendido, metas e subprodutos de cada fase. Existem diferentes propostas para Modelos de Ciclo de Vida e a escolha de um ou outro modelo vai depender de características da área de aplicação.

Métodos contém **técnicas** que podem ser classificadas da seguinte forma:

. **técnicas construtivas**, que oferecem meios para se construir o software de forma disciplinada;

. **técnicas normativas**, que estabelecem normas e atributos de qualidade que devem guiar o processo de desenvolvimento;

. **técnicas gerenciais**, que oferecem meios para planejar e controlar o processo de desenvolvimento.

Instrumentos tornam possível a utilização de métodos, suportando atividades específicas do processo de desenvolvimento de software. Instrumentos podem, também, ser classificados em: instrumentos construtivos, instrumentos normativos e instrumentos gerenciais.

Uma **ferramenta** é a implementação computacional de um instrumento, de determinado método.

A partir destas definições, serão apresentados os componentes do ambiente de desenvolvimento de software orientado a objetos TABA_OBJ.

III.3 - Ciclo de Vida para um Ambiente de Desenvolvimento de Software com Orientação a Objetos

Nesta seção apresenta-se uma série de estudos, visando sugerir um modelo de ciclo de vida a ser adotado pelo Ambiente TABA_OBJ.

Com esse objetivo, apresenta-se um resumo de alguns modelos de ciclo de vida propostos na literatura [33], [50], [51], [52] e [53] com breves comentários sobre a adequação destes modelos ao processo de desenvolvimento de software com orientação a objetos.

Finalmente, a partir de uma análise destes modelos, apresenta-se o modelo de ciclo de vida escolhido para ser adotado pelo ambiente TABA_OBJ, justificando-se essa escolha.

Ao se estudar modelos de ciclos de vida existentes na literatura, tentou-se focalizar aqueles que mais se adequassem ao paradigma de objetos. Isto porque o objetivo final é escolher um modelo de ciclo de vida que suporte um método para desenvolvimento de software com orientação a objetos. Neste sentido, verificou-se que os modelos de ciclo de vida tradicionais não são adequados, já que a

orientação a objetos propicia uma maneira diferente de se desenvolver software.

O desenvolvimento de software com orientação a objetos visa alcançar um alto índice de reusabilidade, facilidade de prototipagem rápida e descaracterização das fases do ciclo de vida. Esta descaracterização significa que o processo de desenvolvimento não é seqüencial, com atividades bem definidas para cada fase, mas sim um processo bastante iterativo. Na fase de projeto, por exemplo, pode-se verificar a necessidade de definição de um novo objeto, tendo-se que voltar para a fase de especificação de requisitos para definir esse novo objeto, sendo todo o processo de desenvolvimento realizado através dessas iterações, até que se obtenha o nível de detalhamento desejado.

Além disso, no desenvolvimento de software com orientação a objetos, trabalha-se com os mesmos elementos de abstração. Isto é, trabalha-se com objetos e mensagens, desde a especificação até a implementação final. Portanto, as atividades exercidas em cada fase do ciclo de vida, tais como identificar objetos, definir suas interfaces, mensagens e métodos, são bastante semelhantes, só diferindo no nível de detalhamento. Porém, como esse tipo de desenvolvimento não é seqüencial, pode-se estar em uma fase com um determinado nível de detalhamento (por exemplo, a fase de projeto detalhado ao especificar-se os algoritmos de um determinado objeto) e surgir a necessidade de criar-se um novo objeto, atividade esta correspondente à fase de especificação.

Por outro lado, o resultado de estudos psicológicos sobre a concepção de programas [54], mostra a importância de certos mecanismos cognitivos e, portanto, a necessidade que um sistema favoreça certas características desta atividade. A atividade de transferir conhecimentos de programa, isto é, de reusabilidade é muito importante. Como nem sempre esse conhecimento se adequa perfeitamente a uma

nova situação, são necessários pequenos ajustes ou adaptações, sendo portanto, a alterabilidade outra característica a ser considerada. Finalmente, a atividade de simulação é igualmente muito importante e um ambiente de programação deve suportá-la.

Portanto, reusabilidade, alterabilidade, extensibilidade e facilidade de prototipagem são características que devem ser exploradas no processo de desenvolvimento de software. Como o modelo adotado, por Versões Sucessivas é semelhante ao modelo de prototipagem, onde cada versão gerada pode ser vista como um protótipo, o desenvolvimento de software com orientação a objetos usando o modelo de ciclo de vida de Versões Sucessivas é propício á extensibilidade e alterabilidade, aproximando-se assim, dos mecanismos cognitivos do ser humano.

Já em métodos convencionais como a Análise e Projeto Estruturado, as atividades de cada fase do ciclo de vida são bem determinadas e seqüenciais. Além disto, os elementos de trabalho são distintos a cada fase. Por exemplo, na fase de Análise Estruturada, trabalha-se com Diagramas de Fluxos de Dados (DFDs) , modelando o sistema em torno de processos e dos dados que fluem destes processos. Na fase de Projeto Estruturado [45], modela-se o sistema em Gráficos de Estruturas, com seus módulos, etc. Já na fase de implementação, o sistema será modelado em torno de rotinas e sub-rotinas, parâmetros, etc. Portanto, se adequam perfeitamente três modelos de ciclos de vida. Os modelos ditos fásicos, pois nesses modelos as atividade para cada fase são bem definidas; os modelos baseados em documentos, pois já existem diversos padrões de documentação para esse tipo de desenvolvimento e também os modelos baseados em custos, uma vez que os métodos existentes para cálculo de custos de projetos se baseiam nesses tipos de desenvolvimento mais seqüenciais. No caso da orientação a objetos, é necessário um modelo de ciclo de vida mais flexível.

A seguir serão apresentados alguns dos modelos encontrados na literatura, descrevendo-os rapidamente e serão feitas considerações a respeito de sua utilização em desenvolvimento de software orientado a objetos.

III.3.1. O modelo fásico

Este modelo segmenta o desenvolvimento em um conjunto de atividades sucessivas, onde cada fase requer informações de entrada bem-definidas, utilizando processos bem-definidos e resultando em produtos, também, bem-definidos [50].

Este modelo consiste basicamente das fases de Análise, Projeto, Implementação, Teste e Manutenção. Cada fase é executada com auxílio de métodos, técnicas e ferramentas específicas. Este modelo também é conhecido como modelo em cascata, uma vez que o sistema vai passando de uma fase para outra progressivamente.

A fase de Análise consiste das atividades de planejamento e especificação de requisitos; a fase de Projeto consiste das atividades de Projeto da Arquitetura e Projeto Detalhado; na implementação as atividades são: codificação, depuração, documentação e testes de unidades de código fonte; a fase de testes possui duas atividades: testes de integração e testes de aceitação.

Apesar do modelo parecer bastante simples ele não reflete a maioria dos processos de desenvolvimento de software, em que costuma haver muita sobreposição e interação entre as fases. Entretanto, ele é um modelo válido para casos em que já se desenvolveu sistemas semelhantes e portanto, consegue-se escrever especificações no início do ciclo de vida.

No desenvolvimento de software com orientação a objetos o processo de desenvolvimento é muito iterativo e ainda bastante informal, sendo difícil ter-se fases e

atividades seqüenciais e com entradas e saídas bem definidas.

III.3.2. Modelo enfatizando "marcos", documentos e revisões

São modelos em que se estabelecem marcos, pontos de revisão, documentos padronizados e prazos estabelecidos pela gerência visando melhorar a visibilidade do produto e tornar o processo de desenvolvimento uma atividade mais pública e o produto algo mais tangível [50]. Isso por sua vez, pode resultar numa melhoria da qualidade do produto, aumentar a produtividade dos programadores e melhorar a moral entre os membros da equipe.

Como o desenvolvimento de software orientado a objetos ainda é bastante intuitivo e informal, fica difícil estabelecer marcos e pontos de revisão.

III.3.3. Modelo baseado em Protótipos

Este modelo [50] enfatiza as fontes de requisitos de software e o uso de protótipos, isto é, maquetes de um produto de software. Ao contrário de modelos de simulação, o protótipo incorpora componentes do produto a ser desenvolvido. Um protótipo possui capacidades funcionais limitadas, baixa confiabilidade e desempenho ineficiente. Porém, com ele se é capaz de mostrar características da interface com o usuário como formato de entrada de dados, relatórios, diálogos, etc.. Deste modo é mais fácil alcançar os requisitos do usuário. Além disso, pode-se testar a eficiência de determinados algoritmos, tempo de resposta de periféricos, etc. O uso de protótipos é muito útil no desenvolvimento de sistemas totalmente novos onde, durante a especificação de requisitos já se pode ir esboçando algumas características funcionais e de interface do sistema.

III.3.4. Modelo baseado em Versões Sucessivas

Este modelo é uma extensão do modelo de prototipagem em que um esqueleto inicial do produto é refinado em níveis crescentes de funcionalidade. Porém, nesse modelo, cada versão sucessiva do produto é um sistema que já pode ser utilizado pelo usuário.

Dos modelos apresentados até agora, este é um dos que mais se adequa ao processo de desenvolvimento de software com orientação a objetos. Isto porque pode-se pensar em desenvolvimento de software com orientação a objetos, através de versões, isto é, desenvolve-se uma versão inicial com um núcleo mínimo de objetos e classes e a partir desta versão, vai-se introduzindo funções e/ou características mais sofisticadas ao sistema. Deste modo, só é necessário se preocupar inicialmente com a especificação dos objetos básicos.

DAVIS et al. [51] fizeram um estudo comparativo sobre modelos de ciclos de vida alternativos. Estes modelos possuem características que diferem dos modelos de ciclo de vida mais tradicionais, tais como os descritos acima. Alguns desses modelos se mostraram muito apropriados para o ambiente TABA_OBJ. A seguir apresentamos este modelos, comentando as vantagens e desvantagens de sua utilização no desenvolvimento de software com orientação a objetos.

III.3.5. Prototipagem rápida descartável

Este modelo consiste na construção de uma implementação parcial rápida antes ou durante a fase de especificação de requisitos, para que o usuário experimente e forneça sugestões para os desenvolvedores auxiliando-os na especificação e alterando-a se necessário.

Uma extensão dessa abordagem seria a de se fazer uma seqüência de protótipos para jogar fora até se chegar ao final.

Esta abordagem foge da filosofia de reusabilidade da orientação a objetos, já que uma vez que se faça um protótipo para jogar fora, o produto final será todo refeito sem que classes e objetos do protótipo possam ser aproveitados. Por outro lado, uma vez que se passe a reusar partes do protótipo na versão final, este deixa de ser um protótipo para "jogar fora".

III. 3. 6. Desenvolvimento Incremental

No desenvolvimento incremental, constrói-se uma implementação parcial e aos poucos vai-se incrementando a complexidade do software, com a introdução de novas funções, melhorando o seu desempenho, etc. Deste modo, o usuário acompanha a evolução do produto e fica mais fácil fazer uma avaliação do software. Caso este não esteja de acordo com os requisitos estabelecidos ou o usuário deseje modificá-los, a alteração do software será mais simples.

Muitas propostas existentes para desenvolvimento de software com orientação a objetos existentes, sugerem um desenvolvimento de dentro-para-fora, ou seja, são especificados e projetados os objetos básicos do sistema e, a partir daí, vão se especificando e projetando novos objetos, até que todos os objetos sejam decompostos e se atinja o nível de detalhe desejado. Assim sendo, essas propostas se inserem no processo de desenvolvimento incremental, onde novas facilidades vão sendo incorporadas ao sistema da mesma maneira. Este modelo é muito semelhante ao modelo de Versões Sucessivas descrito acima.

III. 3. 7. Prototipagem Evolutiva

Ao adotar-se este processo de desenvolvimento, assim como nos modelos anteriores, constrói-se uma implementação parcial do sistema de acordo com os requisitos conhecidos. O usuário usa o protótipo com o objetivo de,através desta experiência, entender-se melhor todos os requisitos.

Ao contrário do desenvolvimento incremental, em que todos os requisitos têm que ser, desde o início, bem conhecidos a fim de que se possa selecionar o subconjunto mínimo a ser apresentado, a prototipagem evolutiva é útil nos casos em que os requisitos ainda não estão muito claros. Nesses casos, os protótipos vão sendo construidos a partir dos requisitos conhecidos, para que estes facilitem a identificação dos outros requisitos.

A prototipagem descartável serve para fazer protótipos dos aspectos que ainda não estão bem definidos, de maneira a permitir que os usuários avaliem vantagens e desvantagens. Estas informações possibilitam a alteração dos requisitos iniciais de modo que estes se aproximem das necessidades reais do usuário. Já a prototipagem evolutiva, serve para, partindo dos poucos requisitos que já estão claros, ir-se construindo protótipos, que vão evoluindo gradativamente, até chegar-se ao produto final.

Esse modelo também parece se adequar ao desenvolvimento de software orientado a objetos, principalmente porque como o protótipo é construido rapidamente, o usuário está sempre acompanhando o desenvolvimento, o que permite que sejam avaliados tanto o produto final, como a adequação dos requisitos para o produto. Além disso, nesse modelo, a reusabilidade é um fator que é levado em conta ao longo do desenvolvimento.

O problema deste modelo é que por gerar um protótipo, fatores como confiabilidade, adaptabilidade, manutenibilidade e desempenho não estão assegurados. Assim sendo, sua utilização depende muito do tipo de aplicação a ser desenvolvida.

III.3.8. Software Reusável

Esta abordagem visa reduzir custos de desenvolvimento através da reutilização tanto de código como de projeto ou

partes de projetos de produtos já existentes, em novos sistemas. Entretanto, esta abordagem depende muito de técnicas para que sejam criados componentes de software reusáveis e técnicas e ferramentas para armazenar e recuperar esses componentes reusáveis, além de técnicas de especificação de componentes que ajudem a catalogar e encontrar componentes relevantes.

Construir software com um grau de modularidade tal, que possa ser reusado cada vez reaproveitando mais e criando menos é o objetivo final da orientação a objetos. O alcance pleno deste objetivo ainda está distante, todavia já existem alguns ambientes de programação orientados a objetos, como Smalltalk, Actor e Eiffel que possuem uma biblioteca de classes reusáveis úteis no desenvolvimento de diversas aplicações como editores de texto, gerenciamento de janelas, etc..

III. 3. 9. Síntese Automática de Programas

O termo, síntese automática de programas, é usado para descrever a transformação da especificação de requisitos ou de projeto de alto nível em código de máquina, através do surgimento das linguagens de muito alto nível (VHLL) e de técnicas algoritmicas e baseadas no conhecimento. Com essa abordagem, o tempo de desenvolvimento é bastante reduzido.

Esta técnica não se adequa a um ambiente de desenvolvimento de software orientado a objetos, pelo fato da orientação a objetos ser uma técnica nova e ainda não haver métodos consolidados para especificação e projeto. Assim sendo, torna-se difícil utilizar uma linguagem de muito alto nível adequada.

III. 3. 10. Modelos de Ciclo de Vida para Orientação a Objetos

Além desses modelos de ciclo de vida apresentados, alguns autores, [33] e [47], propuseram novos modelos ou

variações de modelos já existentes, para serem adotados durante o processo de desenvolvimento de software com orientação a objetos. A seguir veremos as propostas destes autores.

O método GOOD

O método GOOD (General Object-Oriented software Development), proposto por Seidewitz [47], tem como objetivo desenvolver software para ser implementado na linguagem ADA. Para tal, ele se baseia na Análise Estruturada e em alterações do Projeto Estruturado, de modo a gerar as estruturas de ADA.

O modelo de ciclo de vida proposto, baseia-se no modelo em cascata, tendo como principais fases a Análise, o Projeto, Implementação e Reutilização. As alterações ocorrem principalmente na fase de Projeto, onde é sugerida uma série de revisões incrementais em diferentes pontos do projeto, como revisão do projeto preliminar e revisão de projeto detalhado. A fase de reutilização tenta aproveitar as construções já existentes em ADA.

Porém, como em ADA não há biblioteca de classes, nem o conceito de herança, tem que se simular as classes para que se possam reusá-las ou até mesmo criar subclasses a partir delas.

O modelo de Agrupamentos ("Cluster Model")

O ambiente de programação Eiffel, desenvolvido por Meyer [33], também propõe um método de desenvolvimento para projeto com orientação a objetos.

Um agrupamento ("cluster"), para Meyer, é definido como um conjunto de classes que estão relacionadas com um objetivo comum, como por exemplo um agrupamento básico (a biblioteca básica do Eiffel) ou um agrupamento de gráficos (um conjunto de classes para gráficos).

A proposta do modelo de ciclo de vida é baseada no modelo em cascata porém intercala fases do desenvolvimento, como projeto e implementação. Uma nova fase é criada, a fase de Generalização, que também é aglutinada com a fase de Validação. O modelo consiste basicamente destas três fases: Especificação, Projeto/Implementação e Validação/Generalização. Como a abordagem "bottom-up" é adotada, aplicam-se estas fases para agrupamentos, que vão de agrupamentos mais genéricos, fornecendo funções utilitárias, para agrupamentos mais específicos à aplicação. Esses agrupamentos vão sendo gerados, através destas três fases, até que se tenha chegado à implementação final. A seguir, veremos uma figura (Figura 15) para ilustrar o modelo de agrupamentos.

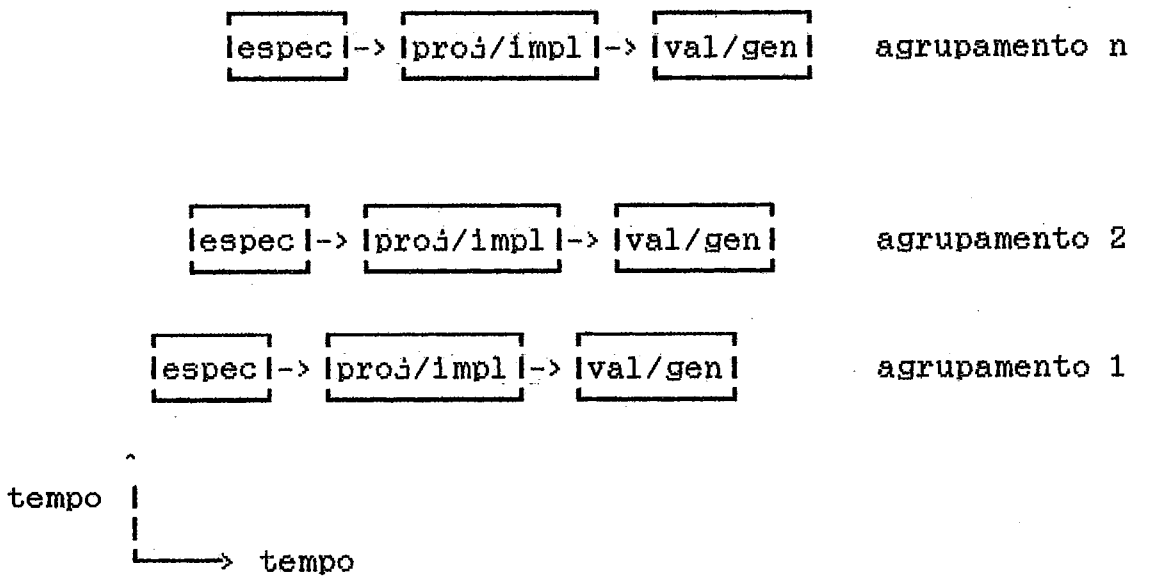

Figura 15 - O Modelo de Agrupamentos

Fonte: MEYER [33]

III.3.11. O Ciclo de Vida do Ambiente TABA_OBJ

Após este estudo dos modelos de ciclo de vida propostos na literatura, concluimos que o modelo de ciclo de vida mais adequado para o Ambiente de Desenvolvimento de Software Orientado a Objetos TABA_OBJ é uma variação dos

modelos de desenvolvimento incremental ou por versões sucessivas. A variação nos modelos de desenvolvimento incremental ou por versões sucessivas, se refere ao procedimento durante a elaboração de cada versão. No desenvolvimento por versões sucessivas, especifica-se inicialmente uma versão com as funções básicas do sistema, projeta-se e finalmente constrói esta versão. A partir daí, novas versões vão sendo construídas aos poucos, introduzindo-se novas funções. Todavia, no desenvolvimento de software com orientação a objetos, mesmo durante o desenvolvimento de cada versão, as atividades de cada fase, por serem muito iterativas, acabam se misturando.

É comum, no desenvolvimento de software com orientação a objetos, durante a fase de projeto, identificarem-se novos objetos, necessitando assim especificá-los. Por outro lado, como o grau de reusabilidade é grande, muitos objetos já existentes poderão ser reaproveitados, não sendo necessário especificá-los e projetá-los. Pode-se também, aproveitar classes já existentes, introduzindo-se um método a mais sendo necessário somente a especificação e projeto desta extensão da classe.

Como não há uma seqüência rígida entre as fases do desenvolvimento e as atividades de cada fase acabam se misturando, pode-se, simultaneamente, estar fazendo o detalhamento do projeto de uma classe e se especificando outra. Uma vez que o desenvolvimento por versões sucessivas parte do desenvolvimento de uma primeira versão bem simplificada e aos poucos vão se construindo novas versões com um grau de complexidade maior, fica mais fácil de se fazer esse tipo de desenvolvimento não sequencial. Para suportar tal estratégia, o ciclo de vida proposto prevê atividades de integração das partes que forem sendo desenvolvidas, e a consequente avaliação da versão resultante desta integração.

Caso fosse se desenvolver um sistema muito grande em uma única versão, seria muito difícil gerenciar os diversos

objetos, cada um em uma fase diferente de desenvolvimento. Por outro lado, como os métodos de desenvolvimento de software com orientação a objetos ainda foram pouco utilizados, torna-se difícil sua utilização para especificar e projetar um sistema todo de uma única vez.

DETIENNE [54] observou que os métodos descendentes, como o método "TOP-DOWN", partem de uma representação abstrata da solução para ir-se detalhando-a aos poucos e são criticados por serem totalmente inadequados às pessoas pouco experientes. Isto ocorre, pois o novato parte de soluções detalhadas e não de representações abstratas para construir um programa.

Por outro lado, o modelo de ciclo de vida adotado, enfatizando a reusabilidade, parece se adequar melhor às capacidades cognitivas do ser humano, acreditando-se, então, que ele vá satisfazer os requisitos desejados.

III.4 - Proposta de um método de desenvolvimento de software com orientação a objetos

A maioria dos métodos de desenvolvimento de software com orientação a objetos existentes cobrem somente parte do ciclo de vida (projeto e construção), sendo sugerido o uso de métodos tradicionais para a fase de Análise de Requisitos. A idéia foi de elaborar-se uma proposta em que a filosofia da orientação a objetos fosse utilizada em todo o ciclo de vida, englobando as fases de Definição, Projeto, Construção e Integração e Avaliação.

A proposta inicial do método foi utilizada em dois projetos, visando avaliar e reformular o método de modo a eliminar as deficiências encontradas. A seguir será descrita a proposta original e a experiência da utilização desta proposta. Finalmente, será apresentado o método revisto após as duas experiências.

III.4.1. Proposta inicial do método [9]

Este trabalho teve por objetivo apresentar um roteiro para desenvolvimento de software, tomando como base os conceitos de orientação a objetos. Sua origem se deu em estudos feitos nesta área, que levaram à motivação de experimentar-se na prática algumas das idéias encontradas na literatura.

O método proposto busca manter uma coerência com o paradigma de objetos e é apresentado, a seguir, através da descrição das fases e atividades que compõem o seu modelo de ciclo de vida.

FASE 1 - Definição

Atividade: Entendimento do problema e definição dos objetivos do produto

Durante esta atividade, deve-se procurar entender todos os aspectos envolvidos no problema a ser resolvido, a fim de se definir os objetivos do produto a ser desenvolvido.

A partir daí, são elaborados os documentos Definição do Sistema e Planejamento Preliminar. Para se fazer o planejamento, deve-se definir estrutura organizacional, cronograma preliminar, métodos de controle da qualidade e usar um método para estimativa de custo.

O método para estimativa de custo adotado é o proposto por LARANJEIRA [55]. Este método consiste de um processo de estimativa de tamanho que posteriormente é relacionado ao custo através do uso do modelo COCOMO [56].

O processo de estimativa de tamanho, cujo modelo funcional é caracterizado por um determinado número de níveis de decomposição dos objetos, pode ser resumido nos seguintes passos:

1) Começando no nível mais baxo de decomposição, calcular o tamanho de cada objeto. Este cálculo deve levar em conta cada função executada pelo objeto assim como suas estruturas de dados;

2) continuar em níveis mais altos, considerando que objetos de nível mais alto podem receber contribuições de objetos componentes, além de seus próprios dados e funções;

3) pode ser necessário incluir objetos "utilitários" para funções de manutenção;

4) o tamanho estimado do sistema como um todo será a soma dos tamanhos estimados dos objetos no nível mais alto de decomposição (que já leva em conta o custos dos objetos dos outros níveis) com o tamanho de possíveis objetos utilitários.

Após o cálculo do tamanho é então utilizado o COCOMO para a estimativa de custo do sistema.

Atividade: Análise e Especificação de Requisitos

A análise de requisitos deve ser feita tomando-se como base uma descrição informal das características do produto que se deseja implementar. Para tal, o texto deve ser examinado de acordo com os passos seguintes:

i) destacar os substantivos existentes buscando-se identificar os objetos e classes, levando-se em conta a ocorrência de substantivos sinônimos que possam representar o mesmo objeto ou classe;

ii) destacar os adjetivos associados aos objetos e classes identificados, buscando-se identificar seus atributos;

iii) destacar os verbos associados aos objetos e classes identificados, buscando-se identificar as operações por estes sofridas ou realizadas.

A especificação dos requisitos deve ser feita tomando-se como base a análise anterior, de acordo com os passos a seguir:

i) reunir os objetos que tenham os mesmos atributos e operações em classes de objetos;

ii) definir a interface das classes de objetos descrevendo suas operações;

iii) verificar a existência de atributos e/ou operações comuns a mais de uma classe de objetos, definindo uma hierarquia entre elas de acordo com o seguinte critério:

- existindo uma classe de objetos cujo conjunto de atributos e operações seja subconjunto dos atributos e operações de uma outra classe qualquer, deve ser definida uma hierarquia de tipos na qual a classe que envolve o conjunto com a quantidade maior de atributos e operações assume o papel de subclasse da outra, sendo portanto desnecessária a definição deste conjunto de características em cada uma delas;

- existindo uma interseção dos conjuntos de atributos e/ou operações de mais de uma classe, deve ser definida uma superclasse que as generalize, passando as classes originais a serem subclasses desta nova classe e sendo, portanto, desnecessária a definição deste conjunto de características comuns em cada uma delas.

Atividade: Planejamento dos ciclos de versões

Durante esta atividade, deve-se definir e planejar a versão a ser construída.

Como o ciclo de vida adotado, baseado em versões sucessivas, sugere que o sistema seja especificado como um todo e projetado por partes, em versões sucessivas, a idéia é que antes de se iniciar o projeto de cada versão, se defina e planeje a versão a ser projetada.

FASE 2 - Projeto

Atividade: Projeto da arquitetura

O projeto da arquitetura deve ser feito tomando-se como base a especificação de requisitos elaborada na fase anterior. Para cada classe de objetos identificada:

i) definir os métodos para as operações identificadas, especificando a sua lógica;

ii) identificar e definir as variáveis de instância, especificando o seu tipo.

Atividade: Escolha do ambiente de programação

O detalhamento do projeto deve ser antecedido pela escolha do ambiente de programação, uma vez que, dependendo do ambiente escolhido, é possível conhecer-se as classes já existentes.

Este conhecimento é importante para a análise de reutilização de classes de objetos feita durante o detalhamento do projeto, já que em função do ambiente de programação a ser adotado, pode existir ou não uma biblioteca de classes disponíveis para reutilização.

Atividade: Detalhamento do projeto

Devem ser detalhados, de acordo com o método proposto a seguir, as classes de objetos já definidas durante o Projeto da Arquitetura.

Tomando-se como base o projeto da arquitetura, para cada classe de objetos:

i) verificar se já existe na biblioteca de classes, alguma classe que possa ser usada para a implementação dos seus objetos;

ii) caso exista, indicar a classe que será usada e as eventuais extensões que possam vir a ser necessárias (definição de novas variáveis de instância e/ou métodos, ou definição de nova subclasse);

iii) caso não exista, decompor a classe de objetos a fim de obter objetos de menor complexidade, e recursivamente projetar a sua arquitetura e detalhar o seu projeto;

iv) representar as trocas de mensagens entre classes de objetos, através de um diagrama.

FASE 3 - Construção

Atividade: Implementação

Implementar os componentes do produto levando em conta os seguintes aspectos:

i) caso seja utilizada uma linguagem de programação orientada a objetos, a implementação resume-se à criação das classes de objetos (variáveis de instância e métodos) no ambiente de programação, indicando sua(s) superclasse(s);

ii) se forem usadas linguagens híbridas ou convencionais, a implementação deve ser feita através da codificação das classes de objetos em um programa propriamente dito; a diferença entre elas é que nas linguagens híbridas existe um tipo de dado que simula classes de objetos reconhecendo, em alguns casos, a herança entre elas. Já nas linguagens convencionais, devem ser criados tipos de dados que simulem as classes de objetos e a herança entre elas.

Atividade: Depuração

Para depurar cada classe de objetos codificada, deve ser simulada a sua execução de acordo com os seguintes passos:

i) substituir as mensagens enviadas a outras classes por uma mensagem externa (exibição para o depurador), e pela inicialização dos parâmetros que estariam contidos na mensagem de volta.

ii) ativar cada uma de suas operações através do envio de mensagens a elas.

Atividade: Planejamento da avaliação do produto

A avaliação do produto deve ser feita em função do processo de integração, isto é, a ferramenta deve ser avaliada à medida que suas partes forem sendo integradas.

Existem duas estratégias para a integração das classes de objetos que compõem o produto. A primeira delas (de-fora-para-dentro) parte dos objetos mais externos (fronteiras do produto) e envolve a integração sucessiva dos objetos mais internos a este. Neste caso, para avaliar cada camada do produto deve ser usada a mesma técnica aplicada à depuração das classes de objetos (ver fase anterior).

A outra alternativa de integração (de-dentro-para-fora) parte dos objetos mais internos (núcleo do produto) e envolve a integração sucessiva dos objetos mais externos a este. Neste caso, para avaliar cada camada do produto, devem ser simuladas as mensagens que entram para as classes de objetos que a compõem.

FASE 4 - Integração e Avaliação

Atividade: Avaliação do produto

Integrar e avaliar o produto de acordo com o definido na fase de planejamento da avaliação do produto.

III.4.2 - Experimentos realizados para Avaliação da Proposta Inicial

O método proposto na seção anterior, foi usado em dois projetos acadêmicos, com o objetivo de permitir sua avaliação e reformulação [10] e [57]. Para tal, seguiu-se o roteiro do método descrito. A partir desta experiência, a proposta original sofreu algumas alterações, visando minimizar as deficiências encontradas durante sua utilização.

Nesta seção, para cada atividade da proposta original, são feitas considerações, a partir da experiência obtida na utilização do método, dando ênfase ao projeto da equipe da qual participei [10].

Este projeto tinha como objetivo desenvolver um protótipo de um Editor Gráfico de apoio ao Projeto Estruturado. Este editor permite se desenhar Gráficos de Estruturas e, já havia sido desenvolvido anteriormente utilizando-se o método de Análise e Projeto Estruturado. Deste modo ficaria mais fácil de se comparar os dois métodos empregados no mesmo projeto.

FASE 1 - Definição

Atividade: Entendimento do Problema e Definição dos Objetivos

No entendimento do problema, a equipe não encontrou grandes dificuldades uma vez que estes projetos basearam-se em outros já existentes e as linhas gerais dos objetivos dos produtos já estavam de alguma forma organizadas em mente.

Atividade: Análise e Especificação de Requisitos

Esta foi a atividade onde mais se observou a necessidade de alteração. Aqui, deveriam-se obter os objetos básicos com suas operações e atributos a partir dos substantivos, verbos e adjetivos de um texto contendo uma descrição informal do projeto.

Tentou-se adotar como descrição informal o texto da Proposta de Desenvolvimento, porém ao destacar-se os substantivos, verbos e adjetivos, ficou-se com uma tabela enorme contendo diversos substantivos repetidos em contextos e níveis de detalhes diferentes, tornando muito difícil a avaliação de quais seriam os objetos básicos relevantes. Além disso a associação de verbos e adjetivos a operações e atributos não foi uma atividade imediata.

Ao tentar escrever uma "descrição informal" que contivesse todos os objetos básicos sem que, por outro lado, possuísse diversos substantivos que não estivessem associados a algum objeto ou que fossem objetos de um nível de detalhamento maior, a equipe encontrou grandes dificuldades.

Além disso, ao discutir-se para ter uma visão global do projeto e de seus elementos (objetos) básicos, a equipe verificou que seria mais simples se o ciclo de vida adotado, baseado em versões sucessivas, ao invés de especificar o sistema como um todo e ir-se projetando em partes, também só especificasse a versão a ser construída. Desta maneira, a etapa inicial se limitaria a especificar um núcleo mínimo de funções necessárias ao sistema, projetá-lo e, aos poucos ir-se especificando extensões a esse núcleo até atingir o produto final. Esse ciclo de vida tem se mostrado bastante adequado à prototipagem rápida e, no caso de um projeto acadêmico, permite que os integrantes da equipe se familiarizem aos poucos com o projeto e com o método, sem deixar de obter resultados práticos.

Uma vez definidos os objetos básicos, a elaboração da Especificação de Requisitos, através da descrição destes objetos básicos e de sua funcionalidade e interface, permitiu uma fácil decomposição inicial desses objetos. No caso do nosso projeto, como o protótipo a ser desenvolvido era pequeno, foi fácil determinar os objetos básicos e não houve a necessidade de efetuar os passos (i), (ii) e (iii) que tentam agrupar em classes os objetos semelhantes.

Sentiu-se, entretanto, falta de um instrumento gráfico que permitisse representar os objetos. Dentre os diagramas apresentados, o diagrama proposto por Coad e Yourdon [6], ainda não conhecido na época da elaboração do método, é o mais completo. Por esta razão, escolheu-se adotá-lo como instrumento de apoio à fase de especificação do método.

Atividade: Planejamento dos Ciclos de Versões

De acordo com o novo ciclo de vida sugerido, esta atividade deveria estar antes da Especificação de Requisitos. No caso do projeto em questão, esta atividade só foi percorrida uma única vez, para se definir o núcleo mínimo do projeto a ser desenvolvido. Porém, como uma das vantagens da orientação a objetos é a extensibilidade, acredita-se que a atividade de planejamento dos ciclos de versões possa ser mantida e facilmente exercida.

FASE 2 - Projeto

Atividade: Projeto da Arquitetura

Nesta atividade foi elaborado o dicionário de classes para a Especificação de Projeto e definidos os algoritmos das funções do Editor Gráfico.

Observou-se que a definição dos métodos e variáveis de instância seria mais adequada na atividade seguinte, onde os algoritmos já estariam detalhados. Além disso, seria mais fácil a reutilização de métodos já existentes, ao invés de ter que defini-los anteriormente, sem saber se poderia haver algum a ser reusado.

Atividade: Escolha do Ambiente de Programação

A escolha do ambiente de programação nesta fase foi bastante vantajosa, pois permitiu que a equipe adiasse esta decisão, até que o projeto já estivesse mais avançado e numa etapa em que a escolha da linguagem de programação definiria o tipo de especificação de métodos e variáveis de instância, etc.

Atividade: Detalhamento do Projeto

Foi feito o detalhamento dos algoritmos, permitindo a visualização/identificação dos métodos necessários.

A representação das trocas de mensagens entre classes através de diagramas se tornou inviável, uma vez que cada

objeto envia ou pode enviar mensagens para diversos objetos simultaneamente e, os novos objetos que sejam criados podem automaticamente receber mensagens de objetos já existentes, evoluindo dinamicamente. Portanto, este diagrama, além de se tornar um emaranhado de arcos ligando objetos,é estático e não acompanha a dinamicidade dos objetos.

As atividades de implementação e depuração foram feitas com o ambiente de programação Actor. Este ambiente tem uma interface muito amigável e sua biblioteca de classes permite um alto grau de reutilização.

III.4.3. O método Revisto

A partir da experiência obtida com os experimentos descritos e tendo em conta as dificuldades descritas na seção anterior, foram feitas algumas modificações no método original que são apresentadas a seguir.

A primeira alteração se refere ao ciclo de vida adotado, baseado em versões sucessivas. Nesta nova proposta, ao invés de se especificar o sistema como um todo e ir-se projetando por partes, vai-se especificando também por partes a cada nova versão.

FASE 1 - Definição

Atividade: Entendimento do problema e definição dos objetivos do produto

Deve-se procurar entender todos os aspectos envolvidos no problema a ser resolvido, a fim de se definirem os objetivos do produto a ser desenvolvido. Para se compreender bem o espaço do problema, deve-se conversar com especialistas, ler sobre o assunto, fazer desenhos ou diagramas, etc.

Atividade: Planejamento dos ciclos de versões

Definir e planejar a versão a ser construída.

Atividade: Análise e especificação de requisitos

A análise de requisitos deve ser feita tomando-se como base uma descrição informal das características do produto que se deseja implementar.

Esta descrição informal tem como objetivo principal a identificação dos objetos básicos e suas operações para a elaboração da versão inicial. Esta versão conterá somente um núcleo mínimo de funções, tornando mais fácil esta identificação. A partir desta versão novos objetos poderão ser facilmente integrados no sistema. A idéia é adotar uma filosofia do tipo "top-down" em que inicialmente são definidos objetos genéricos e, a partir destes, sendo definidas sub-classes com objetos mais específicos.

Não há padrão algum para esta descrição informal, porém, deve-se definir uma estratégia de solução e através de um texto que defina a solução do problema ou de algum diagrama, identificar seus objetos básicos. A partir da identificação dos objetos, deve-se representá-los graficamente junto com suas operações.

A representação gráfica escolhida é um Diagrama de Classes, baseado no diagrama proposto em [6], que foi apresentado na seção II.3.5. Esta escolha foi feita, por acreditar-se que este diagrama é bastante completo e capaz de representar diversas características, como a hierarquia de classes, o todo e suas partes componentes (estrutura de montagem), a cardinalidade do mapeamento entre classes, além da troca de mensagens, atributos e operações das classes.

A especificação dos requisitos deve ser feita tomando-se como base a análise anterior, de acordo com os passos seguintes:

i) reunir os objetos que tenham os mesmos atributos e operações em classes de objetos;

ii) definir a interface das classes de objetos descrevendo as operações que podem ser aplicadas sobre seus objetos;

iii) verificar se uma classe tem componentes ou se é componente de outra;

iv) identificar os atributos de cada classe;

v) definir os requisitos de interface com o usuário das classes;

vi) definir a cardinalidade do mapeamento entre classes (a cardinalidade mínima, define se existindo uma classe, a outra classe associada tem que existir também ou é opcional e, a cardinalidade máxima, que define se para cada classe existe somente um ou M elementos da outra classe);

vii) verificar a existência de atributos e/ou operações comuns a mais de uma classe de objetos, definindo uma hierarquia entre elas de acordo com o seguinte critério:

- existindo uma classe de objetos cujo conjunto de atributos e/ou operações seja subconjunto dos atributos e/ou operações de uma outra classe qualquer, deve ser definida uma hierarquia entre elas de tal forma que a classe que envolva o conjunto com a quantidade maior de atributos e/ou operações assuma o papel de subclasse da outra, herdando desta tais características,

- existindo uma interseção dos conjuntos de atributos e/ou operações de mais de uma classe, deve ser definida uma superclasse que as generalize, passando as classes originais a serem subclasses desta as características comuns.

Uma vez identificados os objetos em classes e subclasses, deve-se elaborar a Especificação de Requisitos e representar as classes de objetos no Diagrama de Classes.

FASE 2 - Projeto

Atividade: Projeto da arquitetura

O projeto da arquitetura deve ser feito tomando-se como base a especificação de requisitos elaborada na fase anterior e, efetuando os seguintes passos:

i) para cada classe de objetos identificada deve-se definir os algoritmos das operações identificadas, especificando a sua lógica. Caso nesse processo sejam identificados novos objetos, deve-se voltar à fase anterior para especificar o novo objeto,

ii) elaborar o dicionário de classes contendo, para cada classe, sua superclasse, suas subclasses, seus componentes, atributos, relacionamentos, operações, funções e objetos especiais da classe (caso hajam);

iii) detalhar os requisitos de interface com o usuário, definindo características mais concretas de como será a interface e como as classes serão representadas.

Esses passos são feitos repetidamente, até que se tenha atingido o nível de detalhamento desejado, especificando e projetando todos os objetos e classes definidos inicialmente e identificados durante o projeto de outros objetos.

Atividade: Escolha do ambiente de programação

A escolha do ambiente de programação deve anteceder ao detalhamento do projeto uma vez que, dependendo do ambiente escolhido, é possível conhecer-se as classes já existentes.

Este conhecimento é importante para a análise de reutilização de classes de objetos feita durante o detalhamento do projeto, já que em função do ambiente de programação a ser adotado, pode existir ou não uma biblioteca de classes disponíveis para serem reutilizadas.

Atividade: Detalhamento do projeto

O detalhamento do projeto deve ser feito tomando-se como base o projeto da arquitetura. Para cada classe de objetos deve-se:

i) verificar se já existe na biblioteca de classes, alguma classe que possa ser usada para a implementação dos seus objetos;

ii) caso exista, indicar a classe que será usada e as eventuais extensões que possam vir a ser necessárias (definição de novas variáveis de instância e/ou métodos, ou definição de nova subclasse);

iii) caso não exista, decompor os seus objetos a fim de obter objetos de menor complexidade e, recursivamente, executar para cada um destes as atividades Projeto da Arquitetura e Detalhamento do Projeto;

iv) representar as trocas de mensagens entre classes de objetos, através de um diagrama.

O diagrama adotado na fase de Análise, representa a troca de mensagens, entretanto, é utilizada uma estratégia para dimunuir o número de mensagens representadas no diagrama. Caso uma classe envie mais de uma mensagem para outra classe, somente um símbolo de envio de mensagem é usado. Caso as subclasses de uma determinada classe enviem mensagens para outra classe, um único símbolo de troca de mensagens é usado na classe e não em cada subclasse.

Desta maneira, pode-se pensar em utilizar o diagrama na fase de projeto detalhado, somente para representar a troca de mensagens, cabendo ao usuário avaliar se o diagrama está muito confuso ou se ele é representativo.

FASE 3 - Construção

Atividade: Implementação

Implementar os componentes do produto considerando que:

i) caso seja utilizada uma linguagem de programação orientada a objetos, a implementação resume-se à criação das classes de objetos (variáveis de instância e métodos) no ambiente de programação, indicando sua(s) superclasse(s);

ii) se forem usadas linguagens híbridas a implementação deve ser feita através da codificação das classes de objetos em um programa propriamente dito utilizando-se o tipo de dado que simula classes de objetos e reconhece, em alguns casos, a herança entre elas;

iii) se forem usadas linguagens convencionais a implementação deve ser feita através da codificação das classes de objetos em um programa propriamente dito criando-se tipos de dados que simulem as classes de objetos e a herança entre elas.

Atividade: Depuração

Para depurar cada classe de objetos codificada, deve ser simulada a sua execução de acordo com os seguintes

passos:

i) substituir as mensagens enviadas a outras classes por uma mensagem externa (exibição para o depurador), e pela inicialização dos parâmetros que estariam contidos na mensagem de volta.

ii) ativar cada uma de suas operações através do envio de mensagens a elas e verificar os resultados obtidos.

Atividade: Planejamento da Avaliação do Produto

A avaliação do produto deve ser feita em função do processo de integração, isto é, o produto deve ser avaliado à medida que suas partes forem sendo integradas.

Existem duas estratégias para a integração das classes de objetos que compõem o produto. A primeira delas (de-fora-para-dentro) parte dos objetos mais externos (fronteiras do produto) e envolve a integração sucessiva dos objetos mais internos a este. Neste caso, para avaliar cada camada do produto deve ser usada a mesma técnica aplicada na atividade Depuração.

A outra alternativa de integração (de-dentro-para-fora) parte dos objetos mais internos (núcleo do produto) e envolve a integração sucessiva dos objetos mais externos a este. Neste caso, para avaliar cada camada do produto, devem ser simuladas as mensagens que são aceitas pelas classes de objetos que a compõem.

FASE 4 - Integração e Avaliação

Atividade: Avaliação do produto

Integrar e avaliar o produto de acordo com o definido na atividade Planejamento da Avaliação do Produto.

O método proposto é resultante da combinação de propostas para projeto e desenvolvimento com orientação a objetos encontrados na literatura. Uma das alterações feitas foi a aplicação, na especificação de requisitos, de algumas técnicas sugeridas para o projeto.

As dificuldades encontradas referiram-se de uma forma geral, à falta de definição de padrões de documentação e técnicas para testar, integrar e avaliar sistemas orientados a objetos.

Acredita-se que apesar de ainda poder ser alterado, este método já corrigiu diversas deficiências encontradas na proposta original. Por outro lado, a existência de um método que cubra todo o ciclo de vida de desenvolvimento de um software, dentro de um ambiente de desenvolvimento de software orientado a objetos, já é um grande passo para que a utilização da orientação a objetos seja cada vez mais difundida e o desenvolvimento de software cada vez mais auxiliado pelo computador.

III.5 - Documentação em um Ambiente de Desenvolvimento de Software com Orientação a Objetos

A documentação durante o processo de desenvolvimento de software é muito importante, por ser uma forma de comunicação entre o desenvolvedor e o usuário final. É através de documentos como a Definição do Sistema e a Especificação de Requisitos, que o usuário tem a oportunidade de dizer se os seus requisitos são exatamente os descritos nessas documentações e deste modo poder participar do processo de desenvolvimento. Além disso, a documentação serve como um histórico de cada passo do processo de desenvolvimento. Esse histórico é bastante útil para as fases de manutenção, alteração e extensão do sistema, pois através dos documentos de um sistema, pode-se verificar como foi feito cada passo do desenvolvimento do sistema, as decisões de projeto, as alterações feitas para versões posteriores e assim por diante.

Uma vez que cada método de desenvolvimento de software tem suas particularidades, os roteiros de documentação variam de acordo com o método adotado para o desenvolvimento.

Em sistemas cujo método de desenvolvimento utilizado é mais orientado a função, como por exemplo a Análise e Projeto Estruturados, os roteiros de documentação enfocam as funções a serem oferecidas pelo sistema, os requisitos funcionais e assim por diante. Já em sistemas com orientação a objetos, a idéia é, desde o início, identificar objetos, encapsulando seus dados e funções. Deste modo, os documentos gerados durante o processo de desenvolvimento, devem descrever esses objetos, apresentando seus requisitos e as características particulares de cada objeto. Por outro lado, é necessário um diagrama que possa representar graficamente os objetos do sistema e sua troca de mensagens. Este Diagrama de Objetos deve fazer parte da Especificação de Requisitos, de modo que o usuário possa visualizar os objetos do sistema e suas interações.

Nesse sentido, foram pesquisados diversos roteiros de documentação para Definição do Sistema, Especificação de Requisitos e Especificação de Projeto. Para a elaboração dos roteiros de documentação para o ambiente TABA_OBJ, procurou-se seguir padrões já existentes, fazendo as adaptações necessárias para atender às particularidades da orientação a objetos.

III.5.1 - Documentos a serem gerados segundo o Ciclo de Vida proposto

A seguir serão apresentados, para cada fase do ciclo de vida, os roteiros de documentação estudados e os roteiros de documetação do TABA_OBJ.

III.5.1.1 - Definição do Sistema

A fase de definição e entendimeto do problema tem como resultado um documento que é a Definição do Sistema. O objetivo da Definição do Sistema é ser o documento inicial em um projeto. Assim sendo, deve descrever a necessidade de se desenvolver um produto, os objetivos que ele visa alcançar e fazer um estudo da viabilidade de sua implementação e operação.

Por ser o primeiro de uma série de documentos que serão gerados durante o desenvolvimento de um produto, a Definição do Sistema é importante pois dá uma visão inicial do que será o sistema de modo geral e também possibilita a elaboração de um planejamento do projeto.

Para se chegar à elaboração do roteiro para a Definição do Sistema do ambiente TABA_OBJ, foram analisados cinco roteiros propostos por: FAIRLEY [50], PRESSMAN [46], Staa (citado por ROCHA [58]), SHOOMAN [59] e ANSI/IEEE [60]. A partir de uma comparação entre essas propostas, foi elaborado o roteiro final.

III.5.1.1.1 - Roteiros propostos na literatura

Para FAIRLEY [50], a definição do sistema deve ser feita em dois documentos. O primeiro é a **Definição do Sistema** propriamente dita, que se concentra mais nos objetivos do sistema, funções oferecidas, características do usuário, etc. O segundo documento, o **Plano do Projeto**, enfoca mais os detalhes operacionais do desenvolvimento do projeto tais como sua estrutura organizacional, o modelo do ciclo de vida adotado, etc. Seguem-se os roteiros propostos para estes dois documentos:

Definição do Sistema

1 - Definição do problema

2 - Justificativa do sistema

3 - Objetivos para o sistema e para o projeto

4 - Restrições no sistema e no projeto

5 - Funções a serem oferecidas (hardware, software, pessoal)

6 - Características do usuário

7 - Ambientes de Desenvolvimento/Operação/Manutenção

8 - Estratégia de solução

9 - Prioridade para as características do sistema

10 - Critério de aceitação do sistema

11 - Fontes de Informação (Bibliografia)

12 - Glossário de Termos

Plano do Projeto

1 - Modelo do Ciclo de Vida (Terminologia, marcos, etc.)

2 - Estrutura Organizacional (estrutura gerencial, equipe)

3 - Equipe Preliminar e Requisitos de Recursos

4 - Planejamento/Escalonameto do desenvolvimento preliminar

5 - Estimativa de Custos Preliminar

6 - Monitoração do Projeto e Mecanismos de Controle

7 - Ferramentas e Técnicas a serem usadas

8 - Linguagens de Programação

9 - Requisitos de Testes

10 - Documentos de Manutenção requisitados

11 - Forma de demonstração e entrega

12 - Tarefa de Treinamento e Material

13 - Plano de Instalação

14 - Considerações de Manutenção

15 - Métodos e tempo de entrega

16 - Métodos e forma de pagamento

17 - Fontes de informação

O objetivo do **Plano do Projeto** para PRESSMAN [46] é comunicar o escopo e os recursos para a gerência, equipe técnica e clientes bem como definir o custo e planos para revisão da gerência, fornecendo uma abordagem geral do desenvolvimento do software para todas as pessoas associadas ao projeto.

No roteiro, tanto o capítulo de Custos como o de Planos, vão variar no nível de detalhamento de acordo com o público alvo do documento. Se for usado como documento interno, o resultado de cada técnica de estimativa de custo deve ser apresentada, mas caso seja enviado para os usuários (clientes), basta apresentar o resultado global das técnicas para estimativa de custos. Segue-se o roteiro do documento proposto por Pressman.

1 - Escopo

   a. Objetivos do projeto

   b. Principais funções

   c. Outras características

   d. Roteiro de desenvolvimento

2 - Recursos

   a. Recursos Humanos

b. Recursos de Hardware

c. Recursos de Software

3 - Custos

4 - Planejamento

a. Redes de tarefas

b. Diagramas de Gantt

c. Tabela de Tarefas/Recursos

A Proposta de Desenvolvimento, de acordo com Staa (citado em ROCHA [58]), tem como objetivo antecipar respostas a perguntas encontradas durante o desenvolvimento e deve conter:

- uma descrição em linhas gerais do sistema atual e suas deficiências;

- uma descrição do sistema a ser desenvolvido, definindo seus objetivos, requisitos, as necessidades e expectativas do usuário e porque é necessário e oportuno desenvolver o sistema proposto;

- dados que permitam avaliar a viabilidade do desenvolvimento e da operação do sistema proposto.

O roteiro proposto por Staa é o seguinte:

1-Descrição do Sistema Atual

1.1-Objetivos

1.2-Fluxo de Dados do Sistema Atual

1.3-Usuário (identificação, necessidades e expectativas, etc.)

1.4-Descrição de Tarefas (objetivos, descrição, dados utilizados, origem, resultados, destinos, esforço para executar a tarefa, etc.)

1.5-Dados e Resultados

1.5.1-Entrada (quem coleta, gera, prepara e transcreve, volume e frequência da coleta, etc.)

1.5.2-Arquivos (quem mantém e destrói dados, volume de dados mantidos em arquivos, etc.)

1.5.3-Resultados (quem utiliza, uso feito pelo destinatário, importância para o destinatário)

1.6-Solicitação de Serviços (quais as solicitações conhecidas, quem origina (pessoa,evento,etc.), como, quando e frequência com que são originados, etc.)

1.7-Controles (métodos utilizados para controlar a qualidade: dos dados de entrada, dos dados mantidos em arquivos, dos resultados, métodos usados para medir, registrar e reportar o desempenho do sistema atual)

1.8-Segurança do Sistema Atual (quais as medidas de prevenção contra acidentes, medidas para recuperação de dados em caso de acidente, qual a necessidade de sigilo quanto aos dados mantidos, etc.)

1.9-Importância do Sistema Atual (importância para alcance dos objetivos do usuário e do cliente, prejuízos ou dificuldades causadas pela deficiência do sistema atual)

2 - Descrição do Sistema Proposto

2.1-Objetivos

2.2-Requisitos (de informação, funcionais, operacionais, de qualidade)

2.3-Fluxo de Dados do Sistema Proposto

2.4-Descrição dos Processos

2.5-Dados (quem coletará, preparará e transcreverá dados,etc.)

2.6-Solicitações de Serviço (quem origina, quais são, como e quando são originados)

2.7-Controles

2.8-Segurança (medidas de prevenção contra acidentes requeridas, etc.)

2.9-Expectativas (data de início da operação; custo de desenvolvimento, operação, manutenção; restrições; requisitos de qualidade, etc.)

2.10-Benefícios (estimativa de aumento da produtividade; estimativa de redução de custos de capital, etc.)

3-Viabilidade de Implementação

3.1-Risco Implementacional (grau de inovação tecnológica estimado; existência ou ausência de ferramentas de suporte ao desenvolvimento)

3.2-Esboço do Plano (fases, produtos por fases; marcos para controle; procedimentos de aprovação; metodologia de controle de alterações, etc.)

3.3-Segurança do Desenvolvimento (procedimentos de proteção e recuperação de acidentes; riscos de danos de correntes de divulgação de produtos; identificação de processos e dados sigilosos)

4-Relação do Sistema com o Ambiente

4.1-Singularidade (resumo de outros sistemas semelhantes na empresa e/ou mercado, etc.)

4.2-Integrabilidade (resumo de outros sistemas com os quais o sistema proposto vai se relacionar, compartilhar usuários e/ou dados)

4.3-Benefícios Globais (importância relativa do sistema proposto com os outros sistemas da empresa; benefícios esperados para a empresa; impacto do sistema proposto sobre os outros sistemas da empresa)

A proposta feita por SHOOMAN [59] é a de elaboração de um Documento de Requisitos. Este documento contém algumas das informações geralmente descritas nos roteiros para Proposta de Desenvolvimento. O roteiro do Documento de Requisitos é o seguinte:

0 - Introdução (princípios de organização, resumos para outras seções, guia de notações)

1 - Características do computador

2 - Interfaces de Hardware

3 - Funções de Software

4 - Restrições de tempo (restrições quanto à tempo de resposta)

5 - Restrições de precisão (restrições quanto a precisão da resposta)

6 - Resposta a eventos indesejados

7 - Subconjuntos (no caso de não se implementar todo o desejado, qual o subconjunto escolhido)

8 - Características Fundamentais (quais as características do sistema que tem que ser mantidas no caso de alguma alteração)

9 - Alterações (tipos de alterações previstas)

10 - Glossário

11 - Fontes (referências)

Outra proposta de roteiro para a **Especificacão de Requisitos de Software** é a do ANSI/IEEE ("American National Standard/ Institute of Electrical and Eletronics Engineers") [60]. O objetivo desta proposta foi gerar um padrão de documentação, descrevendo o conteúdo e as

qualidades desejáveis para uma Especificação de Requisitos de Software (ERS). O roteiro proposto pelo ANSI/IEEE é bastante abrangente e os seus dois capítulos iniciais podem ser perfeitamente utilizados como Definição do Sistema.

O roteiro para este dois capítulos é o seguinte:

1-Introdução

1.1-Objetivo (descreve o objetivo do documento)

1.2-Escopo (identifica o software pelo nome (editor, SGBD, etc.), explica o que o produto fará ou não fará, descreve a aplicação do software (benefícios, objetivos e metas))

1.3-Glossário (definição dos termos, iniciais e abreviaturas usadas nesse documento)

1.4-Referências

1.5-Visão Geral

2-Descrição Geral

2.1-Perspectiva do Produto (descreve se o produto é totalmente independente ou faz parte de um sistema maior (nesse caso descreve as funções de cada componente do sistema maior e identifica as interfaces), identifica as principais interfaces externas do produto, descreve o hardware e periféricos utilizados)

2.2-Funções do Produto (resumo das funções do produto)

2.3-Características do usuário (descreve as características gerais do usuário final que possam afetar os requisitos específicos)

2.4-Restrições Gerais (descreve políticas regulatórias, limitações de hardware, etc.)

2.5-Hipóteses e Dependências

III.5.1.1.2 - Roteiros para a Fase de Definição do Ambiente TABA_OBJ

Após uma análise dos roteiros selecionados, verificou-se ser interessante a elaboração de dois documentos nessa fase. A razão disto é o fato de nessa fase se discutirem questões bastante distintas. Por um lado, é descrito um problema que justifica o desenvolvimento de um sistema, quais os objetivos funcionais e de desempenho que devem ser alcançados, além de outras questões relacionados aos requisitos do sistema a ser construido. Por outro lado, são discutidas questões mais operacionais, ligadas ao planejamento preliminar do desenvolvimento do sistema. Esta abordagem é semelhante à encontrada em FAIRLEY [50].

A seguir descreveremos o conteúdo dos dois documentos a serem elaborados nesta fase: Definição do Sistema e Planejamento Preliminar.

Definição do Sistema

O propósito da definição do sistema é apresentar o problema existente e a justificativa do desenvolvimento de um sistema para solucioná-lo. Além disso, são definidos os objetivos globais e a descrição geral do produto, onde são definidos o ambiente de uso, características do usuário, etc.

O segundo documento aborda questões de planejamento, tais como cronograma e estrutura organizacional.

1 - Propósito do documento

Descreve o objetivo do documento (pode ser um texto padrão para todos os projetos desenvolvidos usando este roteiro de documentação para Definição do Sistema).

1.1 - Sumário

Contém um resumo dos capítulos e seções presentes no documento.

1.2 - Índice

2 - Introducão

2.1 - Definicão do Problema e Justificativa do Sistema

Define o problema existente e justifica o desenvolvimento de um sistema para solucioná-lo.

2.2 - Objetivos Globais

Descreve os objetivos funcionais e de desempenho do sistema e a aplicação do software (benefícios, objetivos e metas).

2.3 - Descricão do Produto

Faz a identificação do software de acordo com a sua área de aplicação, descreve sua funcionalidade e justifica a presença ou ausência de determinadas facilidades. Compara o produto com outros semelhantes existentes no mercado.

3 - Descricão Geral

3.1- Ambiente de uso

Coloca o produto em perspectiva com outros produtos relacionados. Informa se o produto é independente ou se faz parte de outro sistema ou projeto. Caso faça parte de outro sistema ou projeto, localiza e relaciona o produto entre outros produtos, através de um diagrama que definirá a função de cada produto e as principais interfaces do produto.

3.2 - Funcões do Produto

Faz um resumo das funções oferecidas pelo produto e usa um diagrama para relacionar as funções.

3.3 - Características do usuário

Descreve as características gerais do usuário que possam afetar os requisitos específicos.

3.4 - Restrições, hipóteses e dependências

Descreve as restrições que já existam de antemão como, por exemplo, limitações de hardware. e hipóteses como a possibilidade de se conseguir uma ferramenta para a fase de implementação e que caso não se consiga se terá que adotar outra tática de implementação.

3.5 - Prioridades de Implementação

De acordo com o modelo de ciclo de vida adotado (Versões Sucessivas), esta seção contém o que deverá ser implementado em cada versão.

4 - Referências

Contém todas as referências de outros documentos utlizados para a elaboração da Definição do Sistema.

Anexos

A - Glossário

Planejamento Preliminar

1 - Estrutura Organizacional

Descreve a estrutura gerencial, a estrutura da equipe, o esquema de escalonamento de recursos, alocação de mão de obra, etc.

2 - Cronograma Preliminar

Faz uma estimativa do cronograma do projeto incluindo todas as fases do desenvolvimento. Apresentação gráfica de diagramas de Gantt e PERT.

3 - Estimativa de Custos

Faz uma estimativa de custo do projeto usando o método proposto em [55].

3 - Métodos de Controle

Descreve os métodos a serem usados para o controle da qualidade e métodos para controle do desempenho.

4 - Risco Implementacional

Descreve o grau de inovação tecnológica estimado e a existência/ausência de ferramentas de suporte ao desenvolvimento.

5 - Considerações de Manutenção

Define o tipo de manutenção que será dada ao produto, se esta já faz parte do contrato, etc.

6 - Referências

Anexos

A - Glossário de Termos

III.5.1.2 - Especificação de Requisitos de Software

A Especificação de Requisitos de Software é um documento enfatizando o que o software vai fazer e não como será feito. São descritos todos os requisitos do produto, tais como requisitos de qualidade, requisitos funcionais, requisitos da interface com o usuário e restrições de projeto. Outras informações adicionais são incluidas de acordo com a proposta de roteiro de documentação.

III.5.1.2.1 - Roteiros propostos na literatura

A seguir serão apresentados os roteiros propostos por FAIRLEY [50], ROCHA [58] e ANSI/IEEE [60].

A Especificação de Requisitos de software proposta por Fairley [50] tem como objetivo descrever o ambiente de

processamento, as funções básicas do produto, restrições de desempenho, requisitos de interface, etc. Seu roteiro é apresentado a seguir:

1 - Visão Geral do Produto e resumo

2 - Ambientes de Desenvolvimento/Operação/Manutenção

3 - Interfaces Externas e fluxo de dados

3.1-Formato de relatórios e exibições na Tela

3.2-Resumo dos comandos do usuário

3.3-Nível 0 do Diagrama de Fluxo de Dados

3.4-Origem/Destino dos dados lógicos

3.5-Armazenamento dos dados lógicos

3.6-Dicionário de Dados Lógicos

4 - Especificações Funcionais

5 - Desempenho/Requisitos

6 - Condições de Exceção/Tratamento de exceções

7 - Prioridades de implementação

8 - Extensões e modificações previsíveis

9 - Critérios de Aceitação

9.1-Testes funcionais e de desempenho

9.2-Padrões de documentação

10 - Diretrizes do projeto (dicas e restrições)

11 - Fontes de Informação

12 - Glossário de Termos

Segundo Rocha [58], a **Especificação de Requisitos de Software** deve conter a especificação de requisitos funcionais, de informação, de interface, de desempenho e de

qualidade. O roteiro proposto por Rocha leva em conta o uso da Análise Estruturada e será apresentado a seguir.

1-Introdução

1.1-Propósito

1.2-Escopo

1.3-Referências

1.4-Visão Geral da Especificação

2-Descrição Geral

2.1-Interface do produto com outros produtos ou projetos

2.2-Funções do produto

2.3-Características do usuário

2.4-Itens que podem impor limitações nas opções para o projeto

2.5-Suposições e dependências que afetem o software

3-Requisitos Específicos

3.1-Requisitos de Informação

3.2-Requisitos Funcionais

3.3-Requisitos de Interface

3.4-Requisitos de Desempenho

3.5-Requisitos de Qualidade

3.6-Outros Requisitos

...

4 - Prioridade de implementação

5 - Modificações e melhoramentos previstos

6 - Glossário

O padrão de documentação para a Especificação de Requisitos de Software proposto pelo IEEE [60] é dividido em três capítulos, sendo que o terceiro é que trata da especificação de requisitos propriamente dita. Os dois capítulos iniciais se encaixam melhor na fase de definição do sistema e foram apresentados na seção anterior. Para o terceiro capítulo o IEEE apresenta quatro maneiras de classificar os requisitos específicos. A seguir, será vista uma dessas variações.

1-Introdução

1.1-Objetivo do documento

1.2-Escopo

2-Requisitos Específicos

2.1-Requisitos Funcionais

2.1.1-Requisito Funcional 1

2.1.1.1-Introdução

2.1.1.2-Entradas

2.1.1.3-Processamento

2.1.1.4-Saídas

2.1.2-Requisito Funcional 2

...

2.1.n-Requisito funcional n

2.2-Requisitos de Interface Externa

2.2.1-Interface com o Usuário

2.2.2-Interfaces de hardware

2.2.3-Interfaces de software

2.2.4-Interfaces de comunicação

2.3-Requisitos de Desempenho

2.4-Restrições de projeto

2.4.1-Obediência a padrões

2.4.2-Limitações de hardware

...

2.5-Atributos

2.5.1-Segurança

2.5.2-Manutenibilidade

...

2.6-Outros requisitos

2.6.1-Banco de Dados

2.6.2-Operações

2.6.3-Adaptação ao ambiente

...

III.5.1.2.2 - Roteiro para a Especificação de Requisitos de Software do Ambiente TABA_OBJ

Ao se estudar os roteiros propostos na literatura, verificou-se que eles enfatizavam os requisitos funcionais, uma vez que se baseavam em métodos orientados a função. Assim sendo, adaptou-se a estrutura do documento à filosofia de objetos dando ênfase aos requisitos necessários para os objetos básicos do sistema.

Quanto ao diagrama de objetos, após uma pesquisa na literatura, verificou-se que o diagrama de objetos proposto por COAD e YOURDON [6] é bastante completo e adequado ao método proposto para o TABA_OBJ. O diagrama, que já foi apresentado no capítulo II e detalhado na seção III.4.3, fará parte de um anexo do roteiro proposto para a Especificação de Requisitos do TABA_OBJ. A seguir é apresentado o roteiro para a Especificação de Requisitos.

Especificacão de Requisitos

1-Introducão

1.1-Propósito

Esbocar o propósito desta ERS em particular. Especificar o público a quem ela é dirigida.

1.2-Escopo

Identificar o produto pelo nome (ex. Banco de Dados).

1.3-Visão Geral da Especificacão

2-Requisitos Específicos

2.1-Requisitos de Classe 1

2.1.1-Especificacão da Classe 1

Especificacão de <nomeClasse>

Superclasse: <nomeSuperclasse>

Descricão da classe: .....

.....

.....

[Subclasses: { <nomeSubclasse> :

<descrição> }...]

[Contém: { <componente> : [<restricões>,]
<descrição>} ...]

[É componente de: { <classe> :
<descrição> }...]

[Atributos: { <nomeAtributo> : <descrição>,
<restricões> }...]

[Mapeamentos: { com <nomeClasse>,
<multiplicidade>, <ocorrência> }...]

[Métodos: { <nomeMétodo> : <parâmetros>,
<descrição>, <requisitoDesempenho> }...]

[Mensagens Enviadas: { <nomeClasse> :
<parâmetros>, <descrição> }...]

2.1.2-Requisitos de Interface com o Usuário

2.1.2.1-Representacão da classe

2.1.2.2-Acões da Interface com o Usuário

[Acões: { <nomeAcão> : <descricão> <requisitoDesempenho> }...]

2.1.3-Objetos Especiais

2.2-Requisitos de Classe 2

...

2.n-Requisitos de Classe n

3-Prioridade de Implementacão

4-Modificacões e Melhoramentos previstos

5-Glossário

6-Referências

Anexo A - Diagrama de Objetos

III.5.1.3 - Especificacão de Projeto

Após ter-se estudado métodos para o projeto orientado a objetos e roteiros de documentação para projeto baseados em métodos tradicionais, tirou-se algumas conclusões.

Em primeiro lugar, os roteiros de documentação para Especificação de Projeto existentes, são voltados aos métodos tradicionais orientados a função, como o Projeto Estruturado. Os capítulos destes documentos descrevem a decomposição de módulos e processos concorrentes ou defininem gráficos de estrutura e diagramas de fluxo de dados.

Em segundo lugar, os métodos existentes na literatura para projeto com orientação a objetos, são todos muito iterativos, isto é, o projeto consiste de uma seqüência de passos que é repetida diversas vezes até se atingir o nível de detalhe desejado. Do mesmo modo, o método proposto para

o TABA_OBJ também sugere esse processo iterativo, onde o projeto consiste basicamente da definição dos algoritmos das operações dos objetos identificados na fase de especificação e da identificação de novos objetos.

Deste modo, verificou-se que o mais apropriado seria elaborar-se a Especificação de Projeto baseando-se na Especificação de Requisitos, porém detalhando mais a parte que descreve as operações das classes (os serviços). Na Especificação de Projeto, além de descrever os serviços, deve-se apresentar os seus algoritmos, a ordem certa dos parâmetros, etc. Assim como a Especificação de Requisitos descreve o que o produto fará, a Especificação de Projeto descreve como o produto fará.

Quanto aos requisitos da interface com o usuário, estes também devem estar mais detalhados, definindo-se as características finais da interface com o usuário.

O roteiro para a Especificação de Projeto é apresentado a seguir.

Especificação de Projeto

1-Introdução

1.1-Propósito

1.2-Escopo

1.3-Visão Geral da Especificação

2-Requisitos Específicos

2.1-Requisitos de Classe 1

2.1.1-Especificação da Classe 1

Especificacão de <nomeClasse>

Superclasse: <nomeSuperclasse>

Descrição da classe: .....

.....

.....

[Subclasses: { <nomeSubclasse> : <descrição> }...]

[Contém: { <componente> : [<restrições>,] <descrição>} ...]

[É componente de: { <classe> : <descrição> }...]

[Atributos: { <nomeAtributo> : <descrição>, <restrições> }...]

[Mapeamentos: { com <nomeClasse>, <multiplicidade>, <ocorrência> }...]

2.1.1.1- Algoritmos dos Métodos

[{ <nomeMétodo> : <parâmetros>, <descrição>, <requisitoDesempenho>, <algoritmo detalhado> }...]

[Mensagens Enviadas: { <nomeClasse> : <parâmetros>, <descrição> }...]

2.1.2-Requisitos de Interface com o Usuário

2.1.2.1-Representacão da classe

2.1.2.2-Detalhamento das Ações da Interface com o Usuário

[{ <nomeAção> : <descrição>, <parâmetros>, <requisitoDesempenho>, <algoritmo detalhado> }...]

2.1.3-Objetos Especiais

2.2-Requisitos de Classe 2

...

2.n-Requisitos de Classe n

3-Referências

Os roteiros de documentação propostos são frutos de um estudo de roteiros encontrados na literatura, visando

adaptá-los ao paradigma de objetos para dar suporte ao método de desenvolvimento de software proposto para o TABA_OBJ. Para a versão inicial, que é o escopo desta tese, a preocupação maior foi em definir-se roteiros de documentação para Definição do Sistema, Especificação de Requisitos e Especificação de Projeto. Outros roteiros de documentação tais como Manual do Usuário e Plano de Verificação de Software não foram enfocados por não possuerem nenhuma característica especial em se tratando de se utilizar o paradigma de objetos.

CAPÍTULO IV - O PROTÓTIPO DO TABA_OBJ

Este capítulo apresenta uma ferramenta desenvolvida nesta tese, através de sua documentação. Esta ferramenta é um protótipo de um Editor de Diagrama de Classes que foi desenvolvido utilizando-se o método proposto e foi implementado usando o ambiente de programação Actor. Finalmente, são sugeridas outras ferramentas que possam ser oferecidas pelo ambiente TABA_OBJ.

IV.1 - O protótipo do Editor de Diagrama de Classes (EDC)

Após o uso do método do TABA_OBJ em alguns projetos, verificou-se que a ferramenta que mais fazia falta era a de um diagrama de apoio às fases de análise e projeto. Entretanto, das propostas existentes para esses diagramas, a que representa mais informações é o diagrama proposto por [6]. Deste modo optou-se por adotar esse diagrama como um instrumento de representação gráfica das classes na fase de análise e, para viabilizar o uso deste diagrama, implementou-se então um Editor Gráfico de Diagramas de Classes.

A justificativa para o desenvolvimento deste Editor, assim como seus objetivos e principais funções são apresentados no primeiro documento, a **Definição do Sistema**. A identificação das principais classes do sistema e seus requisitos, são vistos no segundo documento, a **Especificação de Requisitos** e o detalhamento destes requisitos e projeto da interface com o usuário são vistos no último documento, a Especificação de Projeto. A seguir serão apresentados esses documentos, que foram feitos baseados nos roteiros propostos nas seções III.5.1.1.2, III.5.1.2.2 e III.5.1.3.

IV.1.1 - Definicão Do Sistema

O roteiro do documento Definicão do Sistema proposto para o TABA_OBJ, consiste de três capítulos e um anexo contendo o glossário (opcional). No primeiro capítulo é feita a introducão, no segundo a definicão do problema e no terceiro, a descricão geral.

A seguir será apresentada a Definicão do Sistema do EDC, de acordo com o roteiro proposto.

Definicão do Sistema do EDC

1. Introducão

1.1 Propósito do Documento

Este documento descreve objetivos e características gerais de uma ferramenta automatizada de software, o **Editor de Diagramas de Classe** (EDC). Esta ferramenta apoia a fase de análise no desenvolvimento orientado a objetos de sistemas.

O EDC permite a descricão das propriedades das classes de objetos identificados durante a fase de análise no desenvolvimento de software com orientacão a objetos. Isto é realizado através da criacão e alteracão interativa de diagrama de classes, seguindo a proposta de Coad e Yourdon [6]. Este editor é uma das ferramentas que compõem o ambiente TABA_OBJ.

1.2. Sumário

No capítulo 2, deste documento, são apresentadas as justificativas para o desenvolvimento deste sistema, descrevendo seus objetivos globais e comparando-o com outros sistemas semelhantes. O capítulo 3 contém a descricão geral do sistema, indicando seu relacionamento com outros sistemas, as principais funcões do produto, as características dos usuários e as dependências, hipóteses e restricões pré-existentes ao desenvolvimento do sistema.

2. Definição do Problema

2.1 Justificativa do Produto

Com o grande sucesso da orientação a objetos, têm surgido inúmeras propostas de desenvolvimento de software com orientação a objetos, no entanto, pouquíssimas sugerem instrumentos como, por exemplo, diagramas de apoio às fases de análise ou projeto. Uma das propostas existentes para esses instrumentos,é a de um diagrama, proposto por Coad e Yordon [6], para o método OOA (Análise com Orientação a Objetos). Por este diagrama ser mais completo que os outros apresentados até então, decidiu-se adotá-lo como instrumento de apoio à fase de análise do método proposto em [10], para o TABA_OBJ. Entretanto, não há ainda uma ferramenta automatizada para a construção desses diagramas.

2.2 Objetivos Globais

A motivação para o desenvolvimento deste sistema é fornecer apoio automatizado para a criação e alteração do Diagrama de Classes [6], viabilizando seu uso e fazendo parte das ferramentas do ambiente TABA_OBJ.

Todavia, o escopo deste trabalho limita-se à elaboração de um protótipo deste Editor de Diagrama de Classes.

2.3 Area de Aplicação

O produto a ser desenvolvido, pode ser classificado como uma ferramenta de desenvolvimento de software. Entretanto, por não haver ainda uma outra ferramenta para a edição de Diagramas de Classes semelhante no mercado, não se pode compará-la com outras ferramentas semelhantes.

3. Descrição Geral

3.1 Ambiente de Uso

A primeira versão do Editor de Diagrama de Classes (EDC) é um protótipo de uma das ferramentas integrantes do TABA_OBJ. Futuramente, o TABA_OBJ poderá ter diversas

ferramentas, sendo um ambiente totalmente automatizado, que fará parte da Estação TABA.

3.2 Funções do Produto

O EDC realiza as seguintes funções:

1) criação e alteração de diagramas de classes,

   O usuário terá controle sobre a disposição dos elementos dentro de um diagrama.

2) especificação de classes de objetos e suas propriedades;

   O usuário poderá criar, alterar e remover definições de classes e seus atributos, métodos, subclasses, componentes e mapeamentos.

3) impressão de diagramas;

4) impressão de resumos de informações de classes;

5) gerência de arquivamento de diagramas;

6) garantia da integridade;

   O usuário não poderá criar definições inadequadas de classes e de suas propriedades.

7) gerência do relacionamento entre diagramas.

   O usuário terá acesso aos cinco níveis de visualização dos diagramas: assunto, objeto, estrutura, atributos e serviços.

3.3 Características dos Usuários

Os usuários desta versão inicial do EDC, são pesquisadores da área que mesmo que nunca tenham usado um método de desenvolvimento de software com orientação a objetos, já estudaram o assunto ou possuem experiência no desenvolvimento de software com outros tipos de métodos.

3.4 Restrições, Hipóteses e Dependências

Por uma questão de integração de ferramentas, optou-se por implementar as ferramentas desenvolvidas para o projeto TABA, em um mesmo ambiente de programação, o ambiente **Actor**. Deste modo, como o Actor utiliza-se do MS-Windows, todas as ferramentas poderão ter o mesmo padrão de interface, o padrão do MS-Windows. O nosso protótipo será desenvolvido em microcomputadores da linha PC (PC-XT e PC-AT).

3.5 Prioridades de Implementação

A versão inicial do EDC, tem como prioridade se implementar as funções básicas para se criar e alterar diagramas (desenhando objetos e ligações) e para armazená-los e recuperá-los, além de imprimi-los.

Outras funções, como a camada de Assuntos [6], a edição e navegação em diversos níveis do diagrama, não serão implementadas ainda nesta versão. Optou-se por inicialmente permitir a edição em um único nível, que abrangeria todas as camadas, com exceção da camada de assuntos. A integração deste editor com o editor de documentos, permitindo o acesso ao EDC a partir do Editor de Especificação de Requisitos de Software (ERS) e vice-versa, além da geração de um esboço da ERS a partir do seu diagrama de classes, também foi deixada para a implementação em versões futuras.

Anexo A - Glossário

IV.1.2 - Especificação de Requisitos

Nesta seção, apresentaremos a especificação de requisitos do EDC, utilizando o roteiro de documentação proposto na seção anterior.

Especificação de Requisitos do EDC

1-Introdução

1.1-Propósito

Este documento tem como objetivo identificar os requisitos de software do produto a ser desenvolvido, o Editor de Diagrama de Classes (EDC). A partir da Definição do Sistema deste editor, serão identificadas suas classes e estabelecidos seus requisitos.

O público alvo deste documento é a equipe envolvida no projeto TABA.

1.2-Escopo

O EDC, conforme foi visto na Definição do Sistema, é um protótipo de uma ferramenta de apoio ao desenvolvimento de software com orientação a objetos.

1.3-Visão Geral da Especificação

A descrição geral do EDC já foi feita na Definição do Sistema, sendo apresentados seus objetivos globais, suas funções, características do usuário e restrições existentes.

A Especificação de Requisitos, então, a partir deste documento, irá apresentar no capítulo 2, os requisitos específicos do EDC, através da especificação de suas principais, identificando seus atributos, métodos, subclasses e assim por diante. No capítulo 3, são definidas prioridades de implementação e o capítulo 4 descreve melhoramentos e modificações previstas para versões futuras. As referências bibliográficas estão no capítulo 5 e finalmente, o Anexo A contém um diagrama das classes descritas neste documento, sumarizando as classes do protótipo a ser desenvolvido.

2-Requisitos Especificos

2.1-Requisitos de Classe

2.1.1-Especificação da Classe Sessão

Especificacão de Sessão

Superclasse:

Descricão da classe: Uma sessão inicia quando o usuário ativa o Editor e termina quando o usuário encerra a sessão.

Subclasses:

Contém:

É componente de:

Atributos:

Mapeamentos: SistemaCorrente com (0:1) Sistema

Métodos:

IniciarSessão: Cria uma sessão, permitindo a edição de diagramas.

EncerrarSessão: Termina a sessão.

AtivarSistema: (S: um sistema) Define o sistema corrente.

Mensagens Enviadas:

2.1.2-Requisitos de Interface com o Usuário

2.1.2.1-Representação da classe

Uma sessão é representada através de uma JanelaDeSessão.

2.1.2.2-Ações da Interface com o Usuário

Ações:

2.1.3-Objetos Especiais

2.2-Requisitos de Classe

2.2.1-Especificação da Classe Sistema

Especificacão de Sistema

Superclasse:

Descricão da classe: Esta classe contém o sistema a ser desenvolvido, com a especificacão de suas classes e com o relacionamento entre as classes.

Subclasses:

Contém:

Classes: um sistema é um conjunto de (0:N) classes.

Relacionamentos: um sistema é um conjunto de (0:N) relacionamentos entre as classes.

É componente de:

Atributos: Nome : nome de um arquivo DOS.

Mapeamentos:

SessãoAberta com (0:1) Sessão

Armazenado com (0:N) Arquivo

Servicos:

CriarSistema

NomearSistema (N: string): define N como o nome do sistema.

IncluirClasse(C: uma classe):

ExcluirClasse(C: uma classe):

restrições:

pré-condições: C pertence à Classes e possui 0 sub-classes.

pós-condições: C não pertence à Classes

Mensagens Enviadas:

2.2.2-Requisitos de Interface com o Usuário

2.2.2.1-Representação da classe

Um sistema é representado por um Diagrama de Classes.

2.2.2.2-Ações da Interface com o Usuário

Ações:

2.2.3-Objetos Especiais

2.3-Requisitos de Classe

2.3.1-Especificação da Classe CLASSE

Especificacão de CLASSE

Superclasse:

Descricão da classe: é responsável pelas classes do sistema.

Subclasses:

Contém:

É componente de: Sistema

Atributos:

Nome: string

Atributos: conj. de strings

Servicos: conj. de strings

Descricão: texto

Mapeamentos:

SuperClasse com (0:1) Classe

Subclasse com (0:1) Classe

TemRelacionamento com (0:N) Relacionamento

Métodos:

CriarClasse

RemoverClasse

DefinirNomeClasse (N: nome)

AcrescentarAtributo (A: string)

RemoverAtributo (A: string)

AcrescentarMétodo (S: string)

RemoverMétodo(S: string)

AlterarDescrição

Mensage Mensagens Enviadas:

2.3.2-Requisitos de Interface com o Usuário

2.3.2.1-Representação da classe

A classe é representada como um elemento do diagrama e também como uma ficha de classe, com o sumário das informações da classe.

2.3.2.2-Ações da Interface com o Usuário

Ações:

MoverClasse: move símbolo de classe pelo diagrama.

DuplicarClase: duplica símbolo de classe.

ImprimirFicha

2.3.3-Objetos Especiais

2.4-Requisitos de Classe

2.4.1-Especificação da Classe Arquivo

Especificacão de Arquivo

Superclasse:

Descricão da classe: Faz o gerenciamento do armazenamento dos diagramas.

Subclasses:

Contém:

É componente de:

Atributos:

Nome: string

Status(char): diz se o arquivo está fechado ou aberto, etc.

Mapeamentos: EmEdicão com (0: 1) Diagrama

Servicos:

CriaArq

NomeiaArq (N: string)

SalvaArq

CarregaArq (N: string): carrega arquivo N na sessão ativa do Editor

Mensagens Enviadas:

2.4.2-Requisitos de Interface com o Usuário

2.4.2.1-Representação da classe

2.4.2.2-Ações da Interface com o Usuário

Ações:

2.4.3-Objetos Especiais

2.5-Requisitos de Classe

2.5.1-Especificação da Classe Relacionamento

Especificacão de Relacionamento

Superclasse:

Descricão da classe: representa o relacionamento entre as classes.

Subclasses:

Mapeamento: responsável pelo mapeamento (cardinalidade mínima e máxima) de uma classe com outra.

Componente: responsável pelo relacionamento de classe/componentes

Mensagem: responsável pelo relacionamento de troca de mensagens entre classes.

Contém:

É componente de:

Atributos:

Origem (C: classe)

Destino (C': conjunto de classe (1: N))

Mapeamentos:

TemRelacionamento com (1: N) Classe

Serviços:

Mensagens Enviadas:

2.5.2-Requisitos de Interface com o Usuário

2.5.2.1-Representação da classe

Um relacionamento é representado pelas ligações do diagrama, tendo cada uma delas um tipo de representação diferente.

2.5.2.2-Ações da Interface com o Usuário

Ações:

2.5.3-Objetos Especiais

2.6-Requisitos de Classe

2.6.1-Especificação da Classe Mensagem

Especificacão de Mensagem

Superclasse: Relacionamento

Descricão da classe: representa o relacionamento de troca de mensagems entre classes.

Subclasses:

Contém:

É componente de:

Atributos:

Destino (C': uma classe):

PosInicial: (P: ponto) ponto indicando a posição inicial da ligação de mensagem.

PosFinal: (P: ponto) ponto indicando a posição final da ligação de mensagem.

Mapeamentos:

Serviços:

IncluirMensagem (P,P'): incluir ligação de mensagem entre P e P'.

RemoverMensagem (R: rect): remover ligação de mensagem contida no retângulo envolvedor.

Mensagens Enviadas:

2.6.2-Requisitos de Interface com o Usuário

2.6.2.1-Representação da classe

O símbolo de mensagem é representado através de uma linha com 2 setas nas extremidades.

2.6.2.2-Ações da Interface com o Usuário

Ações:

MoverMensagem: move o símbolo de mensagem para outra posição.

2.6.3-Objetos Especiais

2.7-Requisitos de Classe

2.7.1-Especificação da Classe Mapeamento

Especificação de Mapeamento

Superclasse: Relacionamento

Descrição da classe: Representa o mapeamento de uma classe com outra, indicando a cardinalidade mínima e máxima.

Subclasses:

Contém:

É componente de:

Atributos:

Destino (C': uma classe):

PosInicial (P: ponto):

PosFinal (P': ponto):

CardMin1 (C: bool): se C é true, a existência de instâncias, em uma das extemidades da ligação é obrigatória (cardinalidade mínima = 1), senão, a existência não é obrigatória (card. mín. = 0)

CardMax1(C: bool): se C é falso, cardinalidade máxima é 1, senão, card. max. é N.

CardMin2 (C: bool): se C é true, a existência de instâncias, na outra extemidade da ligação é obrigatória (cardinalidade mínima = 1), senão, a existência não é obrigatória (card. mín. = 0)

CardMax2 (C: bool): se C é falso, cardinalidade máxima é 1, senão, card. max. é N.

Mapeamentos:

Serviços:

IncluirMapeamento (P, P', CardMin1, CardMax1, CardMin2, CardMax2)

RemoverMapeamento (R: rect)

AlterarCardinalidade (C: num, D: novo valor da cardinalidade): Alterar a cardinalidade C para o valor D.

Mensagens Enviadas:

2.7.2-Requisitos de Interface com o Usuário

2.7.2.1-Representação da classe

Esta classe é representada pela ligação de mensagem.

2.7.2.2-Ações da Interface com o Usuário

Ações:

2.7.3-Objetos Especiais

2.8-Requisitos de Classe

2.8.1-Especificação da Classe Subclasse

Especificação de Subclasse

Superclasse: Relacionamento

Descrição da classe: Representa o relacionamento de uma classe com suas subclasses.

Subclasses:

Contém:

É componente de:

Atributos:

PosInicial (P: ponto):

PosFinal (P': conj. de pontos): conj. de todos os pontos das subclasse.

Mapeamentos:

Serviços:

IncluirLigação (PosInicial, PosFinal): inclui ligação de classe (em PosInicial) com suas subclasses (no conj. de pontos PosFinal).

RemoverLigação (R: rect):

IncluirSubclasse (P: ponto): incluir mais uma subclasse na ligação (na posição P).

RemoverSubclasse (P: ponto): remove uma subclasse da ligação (na posição P).

Mensagens Enviadas:

2.8.2-Requisitos de Interface com o Usuário

2.8.2.1-Representação da classe

A ligação de subclasse é rerpesentada através de uma linha que parte do pai para uma ou mais linhas (das filhas) ligadas por uma barra transversal com um semi-círculo no meio da barra .

2.8.2.2-Ações da Interface com o Usuário

Acões:

2.8.3-Objetos Especiais

2.9-Requisitos de Classe

2.9.1-Especificação da Classe Componente

Especificacão de Componente

Superclasse: Relacionamento

Descricão da classe: Representa o relacionamento de uma classe com seus componentes.

Subclasses:

Contém:

É componente de:

Atributos:

PosInicial (P: ponto):

PosFinal (P': conj. de pontos): conj. de todos os pontos dos componentes da classe.

Mapeamentos:

Servicos:

IncluirLigação (PosInicial, PosFinal): inclui ligação de classe (em PosInicial) com suas classes componentes (no conj. de pontos PosFinal).

RemoverLigação (R: rect):

IncluirComponente (P: ponto): incluir mais uma classe componente na ligação (na posição P).

RemoverComponente (P: ponto): remove uma classe componente da ligação (na posição P).

Mensagens Enviadas:

2.9.2-Requisitos de Interface com o Usuário

2.9.2.1-Representação da classe

A ligação de componente é representada através de uma linha que parte do pai para uma ou mais linhas (das filhas) ligadas por uma barra transversal com um triângulo no meio da barra .

2.9.2.2-Ações da Interface com o Usuário

Ações:

2.9.3-Objetos Especiais

3-Prioridade de Implementação

4-Modificações e Melhoramentos previstos

5-Glossário

6-ReferÊncias

Anexo A - Diagrama de Objetos

IV.1.3 - Especificação de Projeto

Finalmente será apresentada a Especificação de Projeto do EDC, também seguindo o roteiro de documentação proposto.

Especificação de Projeto do EDC

1-Descrição Geral

O EDC permite a especificação de um sistema através da edição interativa de um Diagrama de Classes. Um diagrama representa um sistema, com suas classes e relacionamentos. O EDC, quando ativo, mostra em uma janela (JanelaDaSessão) o diagrama de um sistema, e permite sua edição. Os diagramas seguem a proposta de Coad e Yourdon. Toda a interface com usuário segue as recomendações para aplicações no Ms-Windows.

2-Especificação dos Objetos da Interface com Usuário

2.1-Especificação da Classe

Especificação de JanelaDaSessão

Representa: Sessão

Superclasse: Janela

Descrição da classe: uma JanelaDaSessão é uma janela de documento padrão do Ms-Windows *("Main Window")*, com rolamento *("scrolling")*, redimensionamento, *zoom*, e demais controles. A função desta janela é exibir diagramas de classe, um de cada vez, e permitir a sua edição.

Subclasses:

Contém:

Menus: uma BarraDeMenus

Conteúdo: uma região retangular do diagrama do sistema corrente

É componente de:

Atributos:

Mapeamentos:

Ações:

AbreSessão

entrada:

saída: uma JanelaDaSessão J

descrição: realizado externamente, através do Ms-Windows, com o usuário ativando o EDC.

pré-condições:

pós-condições: J é aberta, não há sistema corrente

AbreSessãoComSistema

entrada: um Sistema S

saída: uma JanelaDaSessão J

descrição: realizado externamente, através do Ms-Windows, com o usuário ativando um arquivo do sistema.

pré-condições: o sistema S tem que existir.

pós-condições: J é aberta, o sistema corrente, S, é exibido na janela.

funcionamento:

IniciarSessão,

AtivarSistema,

FechaSessão

entrada: uma JanelaDeSessão J

saída:

descrição: realizado através do fechamento da JanelaDe-Sessão pelo Ms-Windows ou da opção Fim no menu Arquivo.

pré- pré-condições:

pós-condições:

funcionamento: se Sistema foi modificado após última operação SalvarSistema então será inciado um diálogo perguntando se o usuário deseja salvar as alterções feitas (padrão do Ms-Windows).

CancelarSessão.

Objetos Especiais:

2.2-Especificação da Classe

Especificacão de Diagrama

Representa: Sistema

Superclasse:

Descricão da classe: A classe Diagrama é responsável pela representacão do Diagrama de Classes, ela contém elementos, que são os símbolos de classe e as ligações representando o relacionamento entre as classes. Esse conjunto de elementos é representado por uma coleção ordenada de elementos.

Subclasses:

Contém: Elementos: coleção de elementos;

É componente de:

Atributos:

Vista: retângulo que representa o que está sendo exibido na janela ("client's area" do Ms-Windows).

Mapeamentos:

Ações:

CriarDiagrama

entrada:

saída:

descrição: cria-se um diagrama através da escolha da opção Novo no Menu de Arquivo, seguindo o padrão Ms-Windows.

pré-condições:

pós-condições:

funcionamento:

CriarSistema (Sistema) ,

AtivarSistema (Sistema) ,

DuplicarDiagrama

entrada:

saída:

descrição: através da opção SALVAR COMO... do menu de Arquivo, seguindo o padrão do Ms-Windows.

pré-condições:

pós-condições:

funcionamento:

NomearSistema;

SalvarSistema (Sistema corrente);

Caso tenha se alterado o sistema após a última operação de Salvar, as alterações só serão gravadas no novo arquivo.

Objetos Especiais:

2.3-Especificação da Classe

Especificacão de CLASSE

Representa: O símbolo de CLASSE representa as classes de objetos do sistema.

Superclasse: Elemento

Descrição da classe: responsável pelo gerenciamento da edição e especificação das classes do diagrama. Tem duas formas de representação. A primeira é como um símbolo do diagrama e a segunda é através de uma ficha que o usuário pode editar contendo informações sobre a classe, como: nome, atributos, métodos e observações. O símbolo da classe é um retângulo arredondado dividido em três partes. A primeira tem o nome, a segunda os atributos e a última, seus métodos, conforme mostra a Figura 10 da seção II.3.5.

No primeiro modo de representação, optou-se por definir uma classe como sendo uma janela "filha" da janela de sessão ("child window" de acordo com o Ms_windows). Nesse caso, o gerenciamento de rolamento do diagrama pela janela, seria todo feito pelo Ms_Windows, que já tem funções de rolamento de janelas. Neste caso, bastaria se chamar essa função.

O segundo modo de representação, é uma ficha a ser preenchida pelo usuário contendo um campo para o nome da classe, outro para seus atributos e um terceiro para seus métodos.

Os primeiros quatro atributos e três métodos serão escritos no símbolo de classe, além de seu nome. Os atributos e métodos restantes, ficarão nesta ficha que será armazenada e poderá ser alterada ao longo do tempo. Caso o usuário tenha preenchido na ficha mais de quatro atributos ou três métodos, o último atributo ou método escrito no símbolo de classe será seguido de reticências (...) para indicar que existem mais atributos ou métodos que o que está sendo exibido. Neste caso, se o usuário desejar, ele pode consultar a ficha para verificar os atributos ou métodos restantes.

Subclasses:

Contém:

É componente de:

Atributos:

Posição: indica a posição do símbolo de classe no diagrama.

Mapeamentos:

Ações:

IncluirElementoClasse

entrada:

saída: uma nova classe C no diagrama.

descrição: através da escolha da opção CLASSE, no Menu Diagrama.

pré-condições:

pós-condições:

funcionamento: ao se escolher esta opção, o usuário "clica" com o mouse a posição onde ele deseja que seja desenhado o símbolo de classe. Em seguida, é aberto um diálogo de Especificação de Classe, pedindo informações sobre a classe, como seu nome, atributos e métodos.

Após preencher este diálogo, um símbolo de classe com as informações obtidas, é desenhado na janela na posição indicada.

MoveClasse

entrada: uma classe C

saída:

descrição: Move a posição de C pelo diagrama.

pré-condições: a classe C pertencer ao diagrama

pós-condições:

funcionamento:

O usuário seleciona a classe e arrasta-a pela janela até a posição desejada.

O símbolo anterior é apagado e desenhado na nova posição.

AlterarDescrição

entrada:

saída:

descrição: o usuário solicita a opção de alterar a ficha de descrição da classe através de sua opção no Menu Classe.

pré-condições:

pós-condições:

funcionamento:

Ao selecionar-se uma dessas opções, o sistema apresenta uma ficha na tela, com campos a serem preenchidos com informações sobre a classe.

RemoverClasse

entrada: uma classe C

saída:

descrição: o usuário seleciona a classe C e escolhe a opção Remover do menu. Se C possui alguma ligação com outra classe, é enviada uma mensagem de erro solicitando que as ligações sejam removidas antes, para garantir a integridade do diagrama. Caso contrário, a classe C é removida do diagrama (é enviada uma mensagem para o diagrama remover C de sua coleção ordenada e a JanelaDeSessão é redesenhada, sem a janela "child" que representava a classe C) .

pré-condições:

pós-condições:

funcionamento:

Ao selecionar-se uma dessas opções, o sistema apresenta uma ficha na tela, com campos a serem preenchidos com informações sobre a classe.

Objetos Especiais:

2.4-Especificação da Classe

Especificacão de Ligacão

Representa: Relacionamento

Superclasse: Elemento

Descrição da classe: Representa o símbolo relacionamento entre as classes. Assim como o símbolo de classe, o símbolo de relacionamento também foi definido como sendo composto por várias janelas do tipo "child", uma para cada segmento da ligação.

Subclasses:

Mapeamento

Componente

Mensagem

Subclasses

Contém:

É componente de:

Atributos:

PosInicial: ponto - ponto indicando origem da ligação.

PosFinal: ponto - ponto indicando o destino da ligação.

Mapeamentos:

TemLigação com (1:N) Classe

Ações:

RemoveLigação

entrada: uma ligação

saída:

descrição: remove o símbolo de ligação contida no retângulo envolvedor.

pré-condições: todo o símbolo de ligação estar contido no retângulo envolvedor.

pós-condições:

funcionamento:

O usuário seleciona a ligação que ele deseja remover.

Redesenha-se a região, apagando a ligação.

Objetos Especiais:

2.5-Especificação da Classe

Especificacão de Subclasse

Representa: relacionamento de classe/subclasse

Superclasse: Ligacão

Descricão da classe: é responsável pela representacão do relacionamento de uma classe com suas subclasses.

Subclasses:

Contém:

É componente de:

Atributos:

PosFinal - conj. de pontos indicando as posições onde serão feitas as ligações com todas as subclasses.

Mapeamentos:

Acões:

CriarLigaSubclasse

entrada:

PosInicial: ponto

PosFinal: ponto.

saída: ligação de classe/subclasse entre os pontos.

descrição: Inclui símbolo de ligação entre uma classe e uma subclasse, a partir de PosInicial para o ponto de PosFinal.

pré-condições: y(PosInicial) < mín. y(PosFinal).

pós-condições:

funcionamento:

Calcular o ponto médio de y(PosInicial) e

y(PosFinal) -> D := y(PosFinal - PosInicial)/2,

D' := ponto (x(PosInicial) ,D) ,

Traçar linha vertical de PosInicial à D',

Traçar linha horizontal do ponto de mín (x(PosInicial) , x(PosFinal)) até máx. (x(PosInicial) , x(PosFinal)) com y=D,

Traçar linha vertical de PosFinal até linha horizontal,

Desenhar semi-círculo em cima da linha horizontal e com o raio vertical em cima da linha vertical que parte de PosInicial.

RemoverLigaSubclasse

entrada: uma ligação de subclasse.

saída:

descrição: remove a ligação de classe/subclasse contida no retângulo R.

pré-condições: a barra horizontal da ligação estar selecionada.

pós-condições:

funcionamento:

O usuário seleciona a barra horizontal da ligação. Para cada subclasse da ligação, é enviada uma mensagem RemoveUmaSubclasse. Depois remove-se a barra horizontal e finalmente a barra vertical que sai da superclasse.

Atualiza o diagrama.

a região selecionada é redesenhada apagando a ligação.

IncluirUmaSubclasse

entrada: um ponto P.

saída: mais um segmento na ligação de subclasse.

descrição: incluir mais uma subclasse (posição P) na ligação. Verifica se já existe a ligação.

pré-condições:

pós-condições:

funcionamento:

Se x(P) < x(origem(barraHorizontal))

então

estende linha horizontal de x(origem(barraHorizontal)) até x(P);

senão

se x(P) > x(destino(barraHorizontal))

então

estende a linha horizontal de x(destino(barra-Horizontal)) até x(P) ;

Faz linha vertical de y(P) até D.

RemverUmaSublasse

entrada: um ponto P

saída:

descrição: remove uma subclasse (posição P) da ligação.

pré-condições: existir mais de uma subclasse na ligação

pós-condições:

funcionamento:

Se x(P) = x(origem(barraHorizontal))

então

redesenha a linha horizontal do novo mín. x(PosFinal) até x(destino(barraHorizontal)) ;

senão

se x(P) = x(destino(barraHorizontal))

então

redesenha a linha horizontal de x(origem(barra-Horizontal)) até novo máx. x(PosFinal).

Remove linha vertical de y(P) até D;

Redesenha região.

Atualiza o diagrama.

Objetos Especiais:

2.6-Especificação da Classe

Especificacão de Mensagem

Representa: a troca de mensagens entre as classes

Superclasse: Ligacão

Descricão da classe: é responsável pela troca de mensagens entre duas classes.

Subclasses:

Contém:

É componente de:

Atributos:

ClasseDestino: nome da classe destino da ligação

Mapeamentos:

Acões:

CriaLigacão

entrada: PosInicial: ponto e PosFinal: ponto

saída:

descrição: faz uma ligação de mensagem entre duas classes, nos pontos fornecidos.

pré-condições:

pós-condições:

funcionamento:

Desenha a ligação de mensagem entre PosInicial e PosFinal.

Atualiza o diagrama.

RemoveLigacão

entrada: PosInicial e PosFinal

saída:

descrição: remove a ligação de mensagem entre duas classes (pontos PosInicial e PosFinal).

pré-condições:

pós-condições:

funcionamento:

Redesenha a região, removendo a ligação,

Atualiza o diagrama.

Objetos Especiais:

2.7-Especificação da Classe

Especificacão de Mapeamento

Representa: o mapeamento entre duas classe (cardinalidade).

Superclasse: Ligacão

Descrição da classe: é responsável pela representação dos dois símbolos de mapeamento entre duas classes, o símbolo de ocorrência, informa se, para uma classe existir, tem que haver uma instância da outra classe ou não (cardinalidade mínima); o símbolo de multiplicidade, informa se, pode haver no máximo uma ou N instâncias de uma classe, caso a outra exista.

Subclasses:

Contém:

Símbolo de Multiplicidade

Símbolo de Ocorrência

É componente de:

Atributos:

ClasseDestino

MultiplOri: Define se é 1 ou N a multiplicidade da classe origem;

MultiplDest: Define se é 1 ou N a multiplicidade da classe destino;

OcorrOri: Define se é 0 ou 1 (obrigatória ou não) a ocorrência de instâncias da classe origem;

OcorrDest: Define se é 0 ou 1 (obrigatória ou não) a ocorrência de instâncias da classe destino;

Mapeamentos:

Ações:

CriaLigação

entrada: PosInicial: ponto e PosFinal: ponto

saída:

descrição: faz uma ligação com os símbolos de mapeamento entre duas classes, nos pontos fornecidos.

pré-condições:

pós-condições:

funcionamento:

Se não houver ligação de componente entre as classes

então

Desenha uma linha entre PosInicial e PosFinal.

Abre-se uma caixa de diálogo, pedindo que o usuário informe os símbolos de ocorrência e multiplicidade da origem e do destino.

Desenha os símbolos de ocorrência e multiplicidade nas extremidades da ligação (tanto na linha como na ligação de componentes, caso haja).

Atualiza o diagrama.

RemoveLigação

entrada: uma ligação selecionada

saída:

descrição: redesenha a região, removendo a ligação,

pré-condições:

pós-condições:

funcionamento:

Se houver ligação de componentes

então

Apaga somente os símbolos de mapeamento, redesenhando a linha da ligação de componentes que estava nesta região.

senão

Remove toda a ligação.

Atualiza diagrama.

Objetos Especiais:

2.8-Especificação da Classe

Especificacão de Componente

Representa: Relacionamento de classe/componentes

Superclasse: Relacionamento

Descricão da classe: Representa o relacionamento de uma classe com seus componentes. A ligação de componente é representada da mesma maneira que a ligação de subclasses, só que ao invés de um semi-círculo, no meio da barra horizontal, há um triângulo. Se a ligação só tiver um componente, a ligação é representada por uma linha ligando os dois pontos com um triângulo no meio.

Subclasses:

Contém:

É componente de:

Atributos:

PosFinal: conj. de pontos P', com todos os pontos dos componentes da classe.

Mapeamentos:

Ações:

CriarLigaComposição:

entrada: PosInicial, PosFinal

saída:

descrição: inclui ligação de classe (em PosInicial) com uma classe componente (em PosFinal).

pré-condições:

pós-condições:

funcionamento:

Inclui a ligação de componentes entre PosInicial e PosFinal. Semelhante ao método IncluirLigaSubclasse.

RemoverLigaComposição:

entrada: uma ligação de composição selecionada.

saída:

descrição: remove a ligação de componentes selecionada.

pré-condições:

pós-condições:

funcionamento:

Semelhante ao de RemoverLigaSubclasse.

IncluirComponente:

entrada: um ponto P

saída:

descrição: incluir mais uma classe componente na ligação (na posição P).

pré-condições: já haver uma ligação de componente com uma classe C.

pós-condições:

funcionamento:

Semelhante ao método IncluirUmaSubclasse.

RemverComponente

entrada: um ponto P

saída:

descrição: remove uma componente (posição P) da ligação.

pré-condições: existir mais de uma classe componente na ligação

pós-condições:

funcionamento:

Se x(P) = x(origem(barraHorizontal))

então redesenha a linha horizontal do novo mín. x(PosFinal) até x(destino(barraHorizontal)),

senão se x(P) = x(destino(barraHorizontal))

então redesenha a linha horizontal de x(origem(barra-Horizontal)) até novo máx. x(PosFinal).

Remove linha vertical de y(P) até D,

Redesenha região.

Atualiza o diagrama.

Objetos Especiais:

3-Referências

IV.2 - Propostas de ferramentas para o TABA_OBJ

Para que o TABA_OBJ se torne um ambiente automatizado é necessário que ele possua um banco de dados para armazenar todas as informações dos diversos projetos desenvolvidos. Todavia, a definição e especificação deste banco de dados foge do escopo desta tese. Futuramente, quando se automatizar o TABA_OBJ, esta questão de extrema importância será analisada detalhadamente.

Além do EDC, o TABA_OBJ deve possuir outras ferramentas que auxiliem o processo de desenvolvimento. Uma ferramenta de grande utilidade para o TABA_OBJ, é um editor genérico de documentação baseado em hipertextos. Este editor possuiria padrões ("templates") dos diversos roteiros de documentação usados e permitiria a navegação por todos os documentos de um determinado projeto, através de técnicas de hipertexto. Além disso, poderia-se percorrer os documentos de todos os projetos desenvolvidos no TABA_OBJ que especificassem uma determinada classe, verificando hipóteses de reutilização.

Uma outra ferramenta interessante seria um gerador automático de esboços de especificações de requisitos, através dos diagramas gerados pelo EDC. Uma vez que no EDC são preenchidas informações sobre as classes, como atributos e métodos, além dele representar os relacionamentos entre as classes, essas informações poderiam ser transformadas no padrão do roteiro de especificação de requisitos proposto para o TABA_OBJ. O documento gerado

poderia ser alterado através do editor de roteiros de documentação. Do mesmo modo, poderia-se também, a partir de uma especificação de requisitos, gerar o seu diagrama de classes correspondentes.

Essas são algumas das ferramentas que poderiam fazer parte do TABA_OBJ. Provavelmente existem outras ferramentas que seriam úteis ao ambiente e que não foram enumeradas aqui, mas espera-se que as sugestões contribuam no processo de automatização do TABA_OBJ.

CAPÍTULO V - CONCLUSÕES

Nesse estudo foram pesquisadas diversas fontes, tentando localizar os aspectos mais importantes da orientação a objetos ao longo do ciclo de vida suportado por um Ambiente de Desenvolvimento de Software.

A conclusão obtida é de que apesar de ainda haver muitas divergências quanto aos conceitos básicos e os requisitos necessários para linguagens suportarem a orientação a objetos, cada vez um número maior de pessoas está utilizando essa abordagem e pesquisando sobre o assunto.

Várias vantagens têm sido apontadas como consequência da orientação a objetos. O fato de se poder modelar o problema em termos de objetos torna possível se diminuir o "gap semântico" (distância entre o problema no mundo real e sua abstração). Além disso, é muito mais simples desenvolver software, sem isolar os dados de seus procedimentos, do que através das abordagens tradicionais em que as operações que atuam sobre uma mesma estrutura de dados estão espalhadas nos diversos módulos do sistema.

A programação orientada a objetos tem sido largamente utilizada. Já existem inúmeras linguagens e extensões de linguagens tradicionais, que suportam a orientação a objetos, sendo SMALLTALK-80 a principal delas. Mas, independentemente da linguagem, com a programação orientada a objetos, a atividade de se programar tem se tornado mais simples e os programas, além de menores, são facilmente extensíveis, reutilizáveis e manuteníveis. Por outro lado, a existência de ambientes de programação como o do SMALLTALK-80 e do Actor, mostram como os conceitos de orientação a objetos propiciam a existência de ferramentas úteis para a fase de implementação.

Em termos de métodos e ambientes de desenvolvimento de software com orientação a objetos, observou-se que as pesquisas ainda não conseguiram gerar um produto comercializável. Quanto às propostas para métodos de desenvolvimento de software, são todas ainda muito informais não havendo um método que cubra todo o ciclo de vida do desenvolvimento de software e que já esteja consolidado. Desses métodos, muito poucos sugerem instrumentos de apoio ao processo de desenvolvimento e ainda não há ferramentas automatizadas que implementem esse instrumentos. Finalmente, há pouquíssima documentação a respeito de ambientes de desenvolvimento de software com orientação a objetos.

Nesse sentido, acredita-se que a proposta feita neste trabalho para o TABA_OBJ, sirva não só como uma forma de se pesquisar uma área nova e que tem crescido bastante nos últimos tempos, mas também como uma proposta mais abrangente, onde são sugeridos um modelo de ciclo de vida, um método, roteiros de documentação, instrumentos e uma ferramenta automatizada de suporte ao método que foi desenvolvida utilizando-se um ambiente de programação totalmente orientado a objetos, o Editor de Diagrama de Classes.

Durante o desenvolvimento desta ferramenta, verificou-se que realmente é mais fácil de se desenvolver software com a filosofia da orientação a objetos e que a programação em linguagens com Actor, permite um alto grau de reusabilidade. Porém, como a linguagem Actor pode ser encarada como uma interface de programação orientada a objetos para o Ms-Windows, é necessário o domínio do esquema de programação do Ms-Windows, baseado em janelas e troca de mensagens entre a janela e a aplicação.

O Editor de Diagrama de Classes ainda não é um produto acabado, porém já oferece algumas facilidades que permitem a experimentação e aperfeiçoamento deste tipo de diagrama. Além disso, por ter sido desenvolvido em uma

linguagem de programação orientada a objetos, permite que sejam incorporadas novas funções.

CAPÍTULO VI - REFERÊNCIAS BIBLIOGRÁFICAS

[1] GOLDBERG, A., *SMALLTALK-80: The Language and its Implementation*, Addison-Wesley, 1984.

[2] MONTE, L. C. M., "A invasão dos objetos", a ser publicado, COPPE/UFRJ, junho 1990.

[3] BOOCH, G., "Object-Oriented Development", IEEE Transactions on Software Engineering, V. 12 No. 2, fevereiro 1986.

[4] LORENSEN, W., "Object-Oriented Design", CRD Software Engineering Guidelines, General Electric Co., 1986.

[5] BAILIN, S. C., "An Object-Oriented Requirements Specification Method", Communications of the ACM, Vol. 32 No. 5, maio 1989.

[6] COAD, P. e YOURDON, E., *Object-Orientes Analysis*, YOURDON PRESS, 1990.

[7] ROTEMBERG, H., *Programação Orientada a Objetos: Um enfoque da Engenharia de Software*, Dissertação de mestrado, PUC/RJ, 1987.

[8] SOUZA, J. M., ROCHA, A. R. C. et al., "Uma Estação de Trabalho para o Engenheiro de Software", a ser publicado, COPPE/UFRJ, 1990.

[9] MATTOSO, A. L. Q. e BLUM, H., "Proposta de Desenvolvimento de Software com Orientação a Objetos", Anais do II Simpósio Brasileiro de Engenharia de Software, Canela, 1988.

[10] MATTOSO, A. L. Q., "Orientação a Objetos: Um método e uma experiência", Anais do XXII Congresso da SUCESU, São Paulo, 1989.

[11] DUFF, C., BLISS, W. et al., *The ACTOR Language Manual*, The Whitewater Group, 1989.

[12] COHEN, A., "Data Abstraction, data encapsulation and Object-Oriented Progamming", SIGPLAN Notices, V19-1, 1984.

[13] COX, B., "Message/Object: An Evolutionary Change", IEEE Software, janeiro 1984.

[14] TAKAHASHI, T., *Introdução a Programação Orientada a Objetos*, III EBAI, 1988.

[15] RENTSCH, T., "Object-Oriented Programming", ACM SIGPLAN Notices, V. 17 No. 9, setembro 1982.

[16] STEFIK, M. & BOBROW, D. G. "Object-Oriented Programming: Themes and Variations", The AI Magazine, Vol. 6 No. 4, 1986.

[17] PASCOE, G. A., "Elements of Object-Oriented Programming", BYTE, agosto 1986.

[18] HALBERT, D. C. e BRIEN P. D., "Classes and Subclasses in the Object-Oriented Programming", IEEE Software, setembro 1987.

[19] KAEHLER, T. e PATTERSON, D., "A Small taste of Smalltalk", BYTE, agosto 1986, pp. 145-159.

[20] TOUATI, H. "Is Ada an Object-Oriented Programming Language?", SIGPLAN Notices, V. 22 No. 5, Maio 1987.

[21] COX, B., "Object-Oriented Programming in C", Unix Review, outubro/novembro 1983.

[22] COX, B., "The Object-Oriented Pre-Compiler", SIGPLAN Notices, V18 No. 1, janeiro 1983.

[23] COX, B. "Object-Oriented Programming: A power tool for software craftsman", Unix Review, fevereiro/marco 1984.

[24] COX, B., "Objects, Icons, and Software-ICs", BYTE, agosto 1986.

[25] COX, B., *Object-Oriented Programming - An Evolutionary Approach*, ADDISON-WESLEY, 1987.

[26] STROUSTRUP, B. "An Overview of C++", SIGPLAN Notices, V. 21 No. 10, outubro 1986.

[27] STROUSTRUP, B. *The C++ Programming Language*, ADDISON-WESLEY, 1986.

[28] STROUSTRUP, B., "Classes: An Abstract Data Type Facility for the C Language", SIGPLAN Notices, V. 17 No. 1, janeiro 1982.

[29] TERRY, C., "Object facilitate modular reusable code", *EDN*, novembro 1989.

[30] MEYER, B., "Eiffel: Programming for reusability and extendibility", *SIGPLAN Notices*, V. 22 No. 2, fevereiro 1987.

[31] MEYER, B., "Reusability: The Case for Object-Oriented Design", *IEEE Software*, marco 1987.

[32] MEYER, B., "The Eiffel Environment", *UNIX REVIEW*, V. 6 No. 8, agosto 1988.

[33] MEYER ,B., "From Structured Programming to Object-Oriented Design: The Road to Eiffel", *IEEE Software*, marco 1987.

[34] BRIEN, P., HALBERT, D., KILIAN, M., "The Trellis Programming Environment", *OOPSLA Proceedings*, 1987.

[35] GOLDBERG, A., *SMALLTALK-80: The interactive programming environment*, Addison-Wesley, 1984.

[36] GOLDBERG, A., "The influence of an Object-Oriented Language on the Programming Environment", International Programming Environment, International Edition, McGRAW HILL, 1986.

[37] BOOCH, G., *Software Engineering with ADA*, BENJAMIN CUMMINGS PUBLISHING, 1983.

[38] ROTEMBERG, H., Staa, A. e Ierusalimschy, R., "Um modelo orientado a objetos para o processo de desenvolvimento", Anais VII Congresso da SBC, 1987.

[39] ROTEMBERG, H., "Orientacão a Objetos: Evolucão sem revolucão", *INFO*, marco 1988.

[40] SHLAER, S. e MELLOR, S.J., *Object-Oriented Analysis: Modelling the World in Data*, YOURDON PRESS, 1988.

[41] MONTE, L.C., "Avaliação da orientação a objetos em Projetos", monografia do curso de Controle da Qualidade, COPPE/UFRJ, 1987.

[42] JACOBSON, I., "Object-Oriented Development in an Industrial Environment", *OOPSLA Proceedings*, 1987.

[43] RINE, D.C., "A common error in the object structure of Object-Oriented Design Methods", *ACM Sigsoft*, V.13 No.4, outubro 1987.

[44] YOURDON, E. e CONSTANTINE, L., *Structured Design*, PRENTICE-HALL, 1979.

[45] GANE, C. e SARSON, T., *Análise Estruturada de Sistemas*, Livros Técnicos e Científicos Editora S.A., 1983.

[46] PRESSMAN, R. S., *Software Engineering: A Practitioner's Approach*, 2nd. Edition, McGRAW HILL, 1987.

[47] SEIDEWITZ, E. e Stark, M., "Towards a general object-oriented ADA lifecycle", Proceedings of the Joint 4th Washington Area ADA Technology, ACM NASA Goddard Space Center, et al., Washington D.C., 1987.

[48] ROCHA, A.R.C., AGUIAR, T.C. e BLASCHEK, J.R. "Ambientes para Desenvolvimento de Software: Definição de termos", Relatório Técnico ES-137/89, COPPE/UFRJ, 1987.

[49] CRISPIM, E. M. H., "Definição de termos em Ambientes para Desenvolvimento de Software", Relatório Técnico ES-175/88, COPPE/UFRJ, 1988.

[50] FAIRLEY, R., *Software Engineering Concepts*, McGRAW HILL, 1985.

[51] DAVIS, A. M., BERSOFF, E. H. e COMER, E. R., "A Strategy for Comparing Alternative Software Development Life Cycle Models", IEEE Transactions on Software Engineering, V. 14 No. 10, outubro 1988.

[52] SEIDEWITZ, Ed, "General Object-Oriented Software Development with ADA: A Life Cycle Approach Case", Goddard Space Flight Center, Technology Conference, abril 1988.

[53] POPE, S. GOLDBERG, A. e DEUTSCH, L. "Object-Oriented Approaches to Software Lifecycle Using the Smalltalk-80 System as a CASE Toolkit", *IEEE*,

[54] DETIENNE, F., "L'approche de l'ergonomie cognitive en programmation informatique", Second International Workshop on Software Engineering and its applications, Toulouse-Franca, dezembro 1989.

[55] LARANJEIRA, L. A., "Software Estimation of Object-Oriented Systems", *IEEE Transactions on Software Engineering*, V. 16 No. 5, maio 1990.

[56] BOEHM, B., *Software Engineering Economics*, Englewood Cliffs, PRENTICE HALL, 1981.

[57] BLUM, H. GONCALVES, L. A., ROSSATTO, M. A., COSTA, L. C. e MATTOSO, M. L. "CPRELOBJ - Um Prototipo de Sistema de Banco de Dados Orientado a Objetos", XV CLAIO - Conferencia Latinoamericana de Informatica, CHILE, julho 1989.

[58] ROCHA, A. R. C., *Análise e Projeto Estruturado de Sistemas*, EDITORA CAMPUS, 1987

[59] SHOOMAN, M., *Software Engineering*, McGRAW HILL, New York, 1983.

[60] ANSI/IEEE Std 830/1984, IEEE Standard Guide to Software Requirements Specifications, 1984.