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

# Técnicas diagramáticas para desenvolvimento de software orientado a objetos

por

Marcelo Hideki Yamaguti

Dissertação submetida como requisito parcial para a obtenção do grau de mestre em Ciência da Computação

Prof. Roberto Tom Price
Orientador

Porto Alegre, setembro de 1993

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

**CIP - CATALOGAÇÃO NA PUBLICAÇÃO**

Yamaguti, Marcelo Hideki

Técnicas diagramáticas para desenvolvimento de software orientado a objetos / Marcelo Hideki Yamaguti. – Porto Alegre: CPGCC da UFRGS, 1993.

147p.: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul, Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, 1993. Orientador: Price, Roberto Tom.

Dissertação: Engenharia de Software, Técnicas Diagramáticas, Diagramas, Editor Diagramático, Desenvolvimento de Software, Orientação a Objetos

Universidade Federal do Rio Grande do Sul
Reitor: Prof. Hélgio Trindade
Pró-Reitor de Pesquisa e Graduação: Prof. Claudio Scherer
Diretor do Instituto de Informática: Prof. Clésio Saraiva dos Santos
Coordenador do CPGCC: Prof. Ricardo Augusto da Luz Reis
Bibliotecária do Instituto de Informática: Celina Leite Miranda

Aos meus pais Matheus e Shizuko.

# AGRADECIMENTOS

Agradeço ao meu orientador, prof. Roberto Tom Price,
pela paciência durante a orientação deste trabalho.

Agradeço aos professores, funcionários e colegas de pós-graduação
que direta ou indiretamente contribuíram
para a realização deste trabalho.

Agradeço a Deus por ter me iluminado e me acompanhado
durante toda esta jornada.

# SUMÁRIO

LISTA DE ABREVIATURAS ........ 11

LISTA DE FIGURAS ........ 13

RESUMO ........ 15

ABSTRACT ........ 17

1 INTRODUÇÃO ........ 19
1.1 Motivação ........ 20
1.2 Estrutura geral do trabalho ........ 21

2 TÉCNICAS DIAGRAMÁTICAS ........ 23
2.1 Técnicas diagramáticas tradicionais ........ 24
2.2 Técnicas diagramáticas estruturadas ........ 24
2.3 Apoio ao desenvolvimento de software ........ 26

3 DESENVOLVIMENTO DE SOFTWARE ORIENTADO A OBJETOS ........ 27
3.1 Conceitos ........ 27
3.2 Metodologias de desenvolvimento de sofware ........ 33
3.3 Metodologias baseadas em decomposição ........ 34
3.4 Mudanças trazidas pela abordagem de OO ........ 37
3.5 Fases do desenvolvimento de software OO ........ 39
3.6 Necessidade de apoio diagramático ........ 42

4 NOTAÇÕES DIAGRAMÁTICAS PARA APOIO AO DESENVOLVIMENTO DE SOFTWARE ORIENTADO A OBJETOS ........ 45
4.1 Diagrama de classes e objetos ........ 46
4.1.1 Notações diagramáticas para representação de classes e objetos ........ 47
4.1.2 Notações diagramáticas para representação de relacionamentos entre classes e objetos ........ 51
4.1.3 Notações diagramáticas complementares ........ 55

INSTITUTO DE INFORMÁTICA
BIBLIOTECA

**4.2 Diagrama de estados** .......... 58
4.2.1 Estados .......... 60
4.2.2 Transições .......... 60
4.2.3 Outras características .......... 62
**4.3 Diagrama de fluxo de dados** .......... 64
4.3.1 Processos .......... 65
4.3.2 Fluxos de dados .......... 68
4.3.3 Terminadores .......... 69
4.3.4 Depósito de dados .......... 69
4.3.5 Subdiagramas de fluxos de dados .......... 71
4.3.6 Especificação de operações .......... 71
**4.4 Diagrama de ação** .......... 73

5 APOIO À METODOLOGIA DE DESENVOLVIMENTO DE SOFTWARE ORIENTADA A OBJETOS .......... 79
**5.1 Etapas básicas de uma metodologia de desenvolvimento de software orientada a objetos** .......... 80
5.1.1 Identificação de possíveis classes e objetos .......... 80
5.1.2 Identificação de relacionamentos entre classes .......... 81
5.1.3 Identificação do "ciclo de vida" de um objeto .......... 82
5.1.4 Definição interna de características das classes .......... 83
5.1.5 Revisão de classes definidas .......... 84
**5.2 Comparação de apoio oferecido entre metodologias existentes** .......... 85
5.2.1 Object Oriented Analysis (OOA) .......... 86
5.2.2 Booch - Object Oriented Design (OOD) .......... 87
5.2.3 Object Modeling Technique (OMT) .......... 88
5.2.4 Meyer - Object-Oriented Software Construction .......... 89
5.2.5 Uniform Object Notation (UON) .......... 90

6 AMBIENTE DE APOIO .......... 93
**6.1 Editores diagramáticos** .......... 93
**6.2 Editores textuais/diagramáticos** .......... 94
**6.3 Dicionário de dados** .......... 95
**6.4 Editor diagramático para apoio às notações propostas** .......... 95

7 CONCLUSÃO ........ 97
7.1 Trabalhos futuros ........ 98

ANEXO 1 Especificação dos requisitos do sistema ........ 99

ANEXO 2 Diagramas de Classes e Objetos para definição do modelo estático do sistema (Análise) ........ 101

ANEXO 3 Diagramas de Classes e Objetos para definição do modelo estático do sistema (Projeto) ........ 111

ANEXO 4 Diagramas de Estado para a definição do modelo dinâmico do sistema ........ 127

ANEXO 5 Diagramas de Fluxo de Dados e Diagramas de Ação para a definição do modelo funcional do sistema ........ 135

BIBLIOGRAFIA ........ 143

# LISTA DE ABREVIATURAS

| | |
|---|---|
| DA | Diagrama de Ação |
| DCO | Diagrama de Classes e Objetos |
| DE | Diagrama de Estados |
| DFD | Diagrama de Fluxo de Dados |
| OMT | *Object Modeling Technique* (metodologia) |
| OO | Orientação a Objetos / Orientado a Objetos |
| OOA | *Object Oriented Analysis* (metodologia) |
| OOD | *Object Oriented Design* (metodologia) |
| UON | *Uniform Object Notation* (notação diagramática) |

# LISTA DE FIGURAS

Figura 3.1 Modelo tradicional do ciclo de vida de software ........................ 33
Figura 3.2 Modelo de um ciclo de vida de software OO ........................ 42
Figura 4.1 Notação básica para representação de uma classe ........................ 47
Figura 4.2 Notação para representação resumida de uma classe ........................ 49
Figura 4.3 Notação para representação de uma classe completa ........................ 50
Figura 4.4 Representação de objetos de uma dada classe ........................ 50
Figura 4.5 Representação do relacionamento de associação ........................ 52
Figura 4.6 Cardinalidade de um relacionamento ........................ 52
Figura 4.7 Representação do relacionamento de agregação ........................ 53
Figura 4.8 Representação do relacionamento de utilização ........................ 54
Figura 4.9 Definição do relacionamento de herança ........................ 55
Figura 4.10 Representação de classe postergada ou abstrata ........................ 55
Figura 4.11 Representação de características completamente definidas ........................ 56
Figura 4.12 Representação de classe genérica ou parametrizável ........................ 57
Figura 4.13 Instanciação de uma classe genérica ........................ 57
Figura 4.14 Exemplo de um DE ........................ 59
Figura 4.15 Símbolos utilizados em diagramas de fluxo de dados ........................ 66
Figura 4.16 Representação de um processo em um DFD ........................ 66
Figura 4.17 Representação de um terminador ........................ 69
Figura 4.18 Representação de um depósito de dados ........................ 70
Figura 4.19 Fluxo para tratamento de objeto ........................ 71
Figura 4.20 Sumário das notações de um DA ........................ 74
Figura 5.1 Gráfico de apoio oferecido aos modelos de um sistema ........................ 85
Figura 5.2 Apoio oferecido por OOA ........................ 86
Figura 5.3 Apoio oferecido por OOD ........................ 87
Figura 5.4 Apoio oferecido por OMT ........................ 88
Figura 5.5 Apoio oferecido por Meyer-Nerson ........................ 89
Figura 5.6 Apoio oferecido por UON ........................ 90

# RESUMO

Este trabalho aborda a efetiva utilização de técnicas diagramáticas para o desenvolvimento de software orientado a objetos durante as fases de análise e projeto de sistemas.

Durante o desenvolvimento de software normalmente as especificações resultantes das fases de análise e projeto possuem uma forma gráfica. A utilização de diagramas no desenvolvimento de software busca facilitar a criação de especificações de um sistema e ao mesmo tempo torná-las mais compreensíveis.

A grande maioria das técnicas diagramáticas que existem atualmente são utilizadas para o apoio ao desenvolvimento de software segundo metodologias fundamentadas no paradigma tradicional de decomposição funcional. Diversas técnicas diagrámaticas foram criadas ou adaptadas a fim de suportar os conceitos deste paradigma, acompanhando a própria evolução do mesmo. Neste contexto, são apresentadas as características básicas de técnicas diagramáticas tradicionais que apoiam a este paradigma.

A partir da introdução dos conceitos de orientação a objetos no desenvolvimento de software, surge a necessidade de criação de novas técnicas diagramáticas ou adaptação de técnicas diagramáticas tradicionais para o suporte adequado ao desenvolvimento de sistemas sob este paradigma.

Neste contexto, são abordados os conceitos envolvidos na orientação a objetos e apresentados os aspectos diferenciais no desenvolvimento de software decorrentes da utilização deste paradigma em contraposição aos paradigmas tradicionais.

São também apresentadas as tarefas específicas realizadas durante o desenvolvimento de software, nas fases de análise e projeto, que estão inseridas no ciclo de vida de um software orientado a objetos.

É proposto um conjunto de notações diagramáticas inter-relacionadas adequado ao apoio de um esquema de etapas básicas para o desenvolvimento de software orientado a objetos, bem como às metodologias já existentes. Durante a descrição destas notações diagramáticas, são apresentadas as suas características individuais, adaptações realizadas para o suporte a orientação a objetos, suas aplicações específicas no desenvolvimento de sistemas e o inter-relacionamento existente.

Finalmente, são definidas as características de recursos e facilidades específicas para o apoio às notações propostas. Dentro dos recursos sugeridos inclui-se a definição da implementação de um editor diagramático que é descrito através das notações sugeridas neste trabalho.

**PALAVRAS-CHAVE**: Técnicas Diagramáticas, Diagramas, Desenvolvimento de Software, Orientação a Objetos, Editor Diagramático

**TITLE:** "DIAGRAMMING TECHNIQUES FOR OBJECT-ORIENTED SOFTWARE DEVELOPMENT"

# ABSTRACT

This work tackles the effective use of diagramming techniques for object-oriented software development during analysis and design phases.

During software development the specifications produced by analysis and design usually take a graphical form. The use of diagrams in software development occurs because designers and analysts like to express themselves that way to turn the specifications more understandable.

Most of diagramming techniques in use nowadays support software development following methodologies based on the conventional functional decomposition paradigm. Various diagramming techniques were created or adapted in order to support the concepts of this paradigm, following its own evolution.

With the introduction of object-oriented concepts new diagramming techniques were created or adapted from conventional methodologies.

This work introduces the concepts of object orientation, as well as, the changes originated from the use of this paradigm in software development. Specific steps related to the analysis and design stages in the object-oriented software life cycle are also presented.

A set of interrelated diagramming techniques for supporting object-oriented software development is presented. The description of these diagramming techniques includes new features, discussion of adaptations for object-oriented techniques, specific applications and uses, and their integration.

Finally, the features of specific resources and facilities for supporting the proposed notations are defined. The description of the implementation of a diagrammatic editor, using the notations presented in this work, is included.

**KEYWORDS:** Diagramming Techniques, Diagrams, Diagrammatic Editor, Software Development, Object-Oriented Paradigm.

# 1 INTRODUÇÃO

A engenharia de software busca identificar soluções para a obtenção de maior produtividade na construção de sistemas de melhor qualidade. Diversas metodologias de desenvolvimento de software foram propostas. Dentro do ciclo de vida de um software cada metodologia busca indicar tarefas específicas com o intuito de se obter sistemas adequados.

Para apoiar as tarefas sugeridas, as próprias metodologias também propõem auxílios (ferramentas) específicos à efetiva produção de software. Uma ferramenta comum sugerida em várias metodologias é a especificação de sistemas através de diagramas. A utilização de notações diagramáticas busca acelerar a criação desta especificação, bem como, torná-la mais compreensível.

Com a evolução do conceito de estruturação de software, notações diagramáticas específicas para o apoio ao desenvolvimento de software sob este paradigma foram criadas e adaptadas para acolher os conceitos do mesmo.

Atualmente metodologias fundamentadas na decomposição estruturada de software possuem um conjunto de notações diagramáticas para a especificação de sistemas. Entretanto, o desenvolvimento de software sob estas metodologias normalmente implica na tradução (conversão) de uma notação diagramática utilizada em uma fase de desenvolvimento para outra notação utilizada na próxima fase devido à disjunção existente entre as fases sugeridas pelas metodologias.

Com o surgimento da orientação a objetos o desenvolvimento de software não sofre tal disjunção, uma vez que todo o desenvolvimento de um sistema é baseado em apenas um conceito: o objeto.

Com a introdução dos conceitos de orientação a objetos no desenvolvimento de software diversas metodologias estão sendo sugeridas e

novamente tais metodologias necessitam de um apoio adequado para a execução de suas tarefas.

Surge novamente a oportunidade do apoio diagramático, agora para permitir o desenvolvimento de software orientado a objetos.

## 1.1 Motivação

Com a ascenção do conceito de orientação a objetos no desenvolvimento de software e o aparecimento de novas metodologias. Abre-se a oportunidade para a definição de novas técnicas diagramáticas ou adaptação de técnicas tradicionais para apoio ao desenvolvimento de sistemas sob o novo paradigma.

Atualmente a orientação a objetos é um aglomerado de conceitos e termos técnicos. Muitas vezes o autor de uma metodologia sob este paradigma utiliza conceitos e termos técnicos que não coincidem com os utilizados por outro autor. Deste modo as metodologias propostas sob este paradigma sugerem tarefas que modelam aspectos de um sistema de modos diferentes, normalmente utilizando notações diagramáticas diversas.

O presente trabalho busca identificar quais os aspectos básicos envolvidos no desenvolvimento de sofware orientado a objetos e propõe técnicas diagramáticas específicas para apoiar a modelagem de tais aspectos.

São identificadas as tarefas básicas a serem executadas durante o desenvolvimento de software orientado a objetos e a efetiva utilização das técnicas propostas. Também são propostas facilidades específicas para apoio às notações sugeridas, incluindo a definição de um editor diagramático específico utilizando-se as próprias notações sugeridas.

## 1.2 Estrutura geral do trabalho

O capítulo 2 descreve as vantagens trazidas pela utilização de técnicas diagramáticas no apoio ao desenvolvimento de software, indicando as características de técnicas diagramáticas utilizadas no apoio a metodologias tradicionais.

O capítulo 3 trata do paradigma de orientação a objetos no desenvolvimento de software. Inicialmente são apresentados os conceitos envolvidos no paradigma e, então, são comparadas as características de algumas metodologias tradicionais com a orientada a objetos, indicando as mudanças trazidas pelo novo paradigma. Neste mesmo capítulo, são apresentadas as fases básicas do desenvolvimento de sofware orientado a objetos, caracterizando as principais tarefas envolvidas.

No capítulo 4 são apresentadas as notações propostas para a modelagem de sistemas orientado a objetos. Para cada notação são apresentadas as características específicas, a utilização e inter-relacionamento com as outras notações diagramáticas.

O capítulo 5 apresenta um esquema de tarefas para o desenvolvimento de software no qual é enfatizada a utilização das notações propostas. Também neste capítulo são comparadas algumas metodologias atualmente existentes com o conjunto de notações diagramáticas proposto.

O capítulo 6 sugere algumas facilidades e recursos necessários para apoiar as notações sugeridas, que inclui um editor diagramático específico cuja especificação é detalhada nos anexos.

# 2 TÉCNICAS DIAGRAMÁTICAS

A utilização de diagramas claros e adequados pode auxiliar o desenvolvimento de sistemas complexos. Do mesmo modo que a utilização de algarismos romanos na matemática pode implicar em uma grande dificuldade na realização de operações aritméticas como a multiplicação e divisão, a utilização de diagramas inadequados na computação pode implicar em limitação no desenvolvimento de um sistema.

O ditado: "uma imagem vale mais que mil palavras", no contexto de notações diagramáticas, é válido apenas se os diagramas produzidos possuírem pelo menos as características de clareza, precisão e concisão. Sob estas características a especificação de sistemas complexos através de diagramas é melhor do que através de textos, que normalmente podem ser confusos e ambíguos.

Uma notação diagramática deve realizar o papel de uma linguagem através da qual um projetista[1] possa criar e especificar um sistema, enquanto que outro projetista possa facilmente ver e entender os diagramas criados pelo primeiro.

Do ponto de vista de um único projetista, a utilização de uma notação diagramática adequada serve como auxílio à clareza de raciocínio, pois, por outro lado, a escolha de uma notação inadequada pode mesmo inibir o trabalho de desenvolvimento.

Do ponto de vista de uma equipe que trabalha no desenvolvimento de um mesmo sistema, uma notação diagramática formal serve como meio de comunicação entre seus projetistas.

---

[1] **Projetista**: analista, projetista ou engenheiro de software que desenvolve o sistema.

## 2.1 Técnicas diagramáticas tradicionais

Técnicas diagramáticas estão em constante desenvolvimento. Isto ocorre para acompanhar a própria evolução de conceitos. Semelhante à variedade de linguagens de programação e métodos de desenvolvimento e especificação de software, diversas notações diagramáticas surgiram para apoiar diferentes tarefas: diagrama de fluxo de dados [DEM 78] [GAN 79], diagrama de estrutura [PAG 88], diagrama entidade-relacionamento [CHE 76], redes de Petri [HEU 91], tabelas e árvores de decisão [MAR 87], diagrama Nassi-Shneiderman [NAS 73], etc.

Técnicas diagramáticas que não suportam construções ou idéias importantes são abandonadas, ou então sofrem adaptações para se atualizarem. Os tradicionais fluxogramas, por exemplo, estão atualmente caindo em desuso pois não refletem construções estruturadas.

Aliás, a introdução de técnicas estruturadas no desenvolvimento de software abriu a oportunidade à criação de diversas notações diagramáticas. Muitas destas notações suportam a abordagem de desenvolvimento estruturado '*top-down*', descrevendo um sistema, ou um programa, em vários níveis de decomposição.

## 2.2 Técnicas diagramáticas estruturadas

Técnicas estruturadas necessitam de notações diagramáticas estruturadas. A utilização de técnicas estruturadas baseadas em notações diagramáticas apresenta as seguintes funções [MAR 87] [MCC 89]:

- *auxílio à clareza de raciocínio*:

A utilização de técnicas diagramáticas apropriadas permite ao projetista maior expressão do seu raciocínio pois poderá visualizar e expressar seu pensamento de forma mais clara. Uma boa escolha da técnica diagramática pode implicar em rapidez e aumento da qualidade dos resultados.

- *comunicação entre os membros da equipe de desenvolvimento*:

No desenvolvimento de sistemas complexos onde está envolvida uma equipe de várias pessoas, os diagramas servem de ferramenta essencial para a comunicação, permitindo aos projetistas trocar idéias, bem como, desenvolver componentes separados que possam ser acoplados convenientemente.

- *documentação de sistemas*:

Técnicas diagramáticas utilizadas durante o desenvolvimento de sistemas tornam os próprios diagramas resultantes como forma de documentação. Os diagramas podem descrever um sistema como especificações de alto nível até detalhes de implementação dos programas.

- *auxílio à depuração*:

Os diagramas que descrevem um sistema facilitam a tarefa dos encarregados da depuração, uma vez que eles refletem a estrutura do sistema. Deste modo se a procura por um determinado erro é necessária, diagramas detalhados podem indicar a sua exata localização; ao passo que se é necessária a procura por determinada função realizada em um conjunto de programas, um diagrama de alto nível é mais adequado.

- *auxílio à manutenção*:

Diagramas tornam claras as estruturas de dados, procedimentos e atividades, deste modo permite-se prever melhor os efeitos que podem ocorrer devido a modificações.

- *rigor nas especificações e verificação automática de erros*:

A utilização de um editor de diagramas automatizado permite que outras facilidades possam ser alcançadas, como o maior rigor nas especificações, uma vez que o editor poderia guiar o projetista na criação de diagramas de acordo com as regras de edição implícitas da notação diagramática utilizada; e também a verificação automática dos erros que possam ocorrer durante a edição de diagramas, liberando o projetista para tarefas intimamente ligadas ao desenvolvimento do sistema.

## 2.3 Apoio ao desenvolvimento de software

Muitas das notações diagramáticas atuais foram criadas para apoiar metodologias que se baseiam no paradigma de desenvolvimento estruturado de software.

Entretanto, as metodologias estruturadas estão evoluindo constantemente. Técnicas estruturadas dos anos 70 vêm sofrendo alterações e novas metodologias vão sendo propostas.

Esta evolução começou com os princípios da programação estruturada (no início dos anos 70), passando-se pela definição de métodos para projeto estruturado (na metade dos anos 70). No final dos anos 70 surgiram as definições de métodos para análise estruturada, bem como de técnicas para definição de banco de dados. Nos anos 80 apareceram técnicas automatizadas, e atualmente existem diversas metodologias baseadas em ferramentas CASE ('*Computer Aided Software Engineering*').

As notações diagramáticas buscam acompanhar esta evolução e deste modo muitas notações foram criadas ou adaptadas para apoiar os diversos métodos e técnicas estruturadas. Com o surgimento do paradigma de orientação a objetos, uma nova gama de conceitos foi introduzida e deste modo surgiu-se a necessidade de um novo conjunto de notações diagramáticas que apoiem adequadamente o desenvolvimento de software sob este novo paradigma.

Os conceitos envolvidos no paradigma de orientação a objetos são apresentados no capítulo seguinte.

# 3 DESENVOLVIMENTO DE SOFTWARE ORIENTADO A OBJETOS

Nos últimos anos a comunidade de software tem testemunhado a ascenção e popularização do paradigma de desenvolvimento de software denominado de "orientação a objetos". Tal paradigma apresenta diversos conceitos que isoladamente não são novos, porém reunidos formam a base deste paradigma que está se tornando o sinônimo computacional de "bom".

A orientação a objetos (OO) possui uma grande variedade de termos técnicos ainda não padronizados. Diversos autores relatam seus trabalhos sobre terminologias e conceitos próprios. Deste modo, para estabilizar a utilização de termos técnicos neste trabalho, inicialmente são apresentados os conceitos envolvidos pelo paradigma, e após são discutidos pontos relativos ao desenvolvimento de software OO.

## 3.1 Conceitos

O termo **"orientação a objetos"** superficialmente significa a organização de software como uma coleção de objetos discretos que incorporam tanto a estrutura de dados como o comportamento [RUM 91].

Um **objeto** é simplesmente algo que faz sentido no contexto de uma aplicação. Um objeto é um conceito, uma abstração, ou coisa com limites e significados claros para um determinado problema. Todo objeto possui uma **identidade** que é a propriedade que o distingue de outros objetos. Além da identidade um objeto também possui um estado e um comportamento [BOO 91].

O **estado** de um objeto engloba todas as propriedades (normalmente estáticas) do objeto mais os valores (normalmente dinâmicos) de cada uma destas propriedades. **Comportamento** é o modo como um objeto age e reage, em termos de suas mudanças de estado e passagem de mensagens.

Um conjunto de objetos que compartilham a mesma estrutura e comportamento podem ser agrupados em uma classe. Uma **classe de objetos** descreve um grupo de objetos com propriedades similares, comportamento, relacionamentos com outros objetos e semântica comum. Cada classe descreve um conjunto infinito de objetos. Cada objeto é dito ser uma instância de sua classe.

Cada classe descreve a estrutura de dados e comportamento de seus objetos, respectivamente por seus atributos e operações. Um **atributo** é um valor de dados guardado pelos objetos de uma classe. Cada atributo dentro de uma classe é identificado por um nome que deve ser único para a classe. Contudo, um mesmo nome de atributo pode ocorrer em classes diferentes. Cada atributo tem um valor em cada instância de objeto; todavia, dois objetos de mesma classe podem possuir o mesmo valor de atributo. Atributos devem ser em princípio valores de dados puros, entretanto, a implementação em algumas linguagens de programação OO permitem definir atributos de uma classe como um objeto de outra classe.

Uma **operação** é uma função ou procedimento que pode ser aplicada sobre ou por objetos em uma classe. Todos os objetos de uma classe compartilham as mesmas operações, deste modo, cada objeto "sabe" como executar suas próprias operações. Uma operação pode possuir opcionalmente argumentos para sua resolução. Um **método** é a implementação específica de uma operação em uma classe.

Uma mesma operação pode ser usada em diferentes classes. Tal operação é considerada então polimórfica. **Polimorfismo** significa a habilidade de tomar várias formas e indica que uma mesma operação pode se comportar diferentemente em diferentes classes. A seleção de determinado método que implementa uma operação em várias classes é determinada pela linguagem de programação OO em tempo de execução. Tal processo é denominado de **ligação dinâmica** [MEY 88] [BOO 91], ou **resolução de método** [RUM 91].

Do ponto de vista da programação OO [REN 82] [STR 88], "classe" é um conceito existente em tempo de compilação e "objeto" é uma instância de uma classe em tempo de execução. O termo "objeto" é muito vago na literatura. A palavra "objeto" em certas ocasiões significa uma única coisa, em outras se refere a um grupo de coisas similares. O contexto normalmente resolve a ambigüidade. Entretanto, neste trabalho o termo "objeto" será utilizado para indicar apenas uma coisa no singular, ao passo que "classe" ou "classe de objetos" indicará o conjunto de objetos de mesma estrutura e comportamento.

Durante a abordagem de orientação a objetos alguns aspectos são levados em consideração [BOO 91] [MEY 88] [RUM 91]:

- **Abstração**

Uma abstração denota as características essenciais de um objeto que o distingue de outros tipos de objetos e deste modo provê limites conceituais bem definidos, relativos à perspectiva de um observador. No desenvolvimento de um sistema isto significa identificar o que um objeto é e o que faz sem decidir antecipadamente como será implementado. A abstração provê a definição da **interface** de uma classe, onde são definidos atributos e métodos disponíveis para outras classes do sistema.

- **Encapsulamento**

É o processo de ocultamento de todos os detalhes de um objeto que não contribuem para suas características essenciais. O encapsulamento provê a **implementação** de um objeto. Na implementação é descrita a representação da abstração bem como os mecanismos para se obter o comportamento desejado. A implementação de um objeto pode ser alterada sem afetar as aplicações que o utilizam. Tais alterações podem ser realizadas para melhoria de desempenho, correção de erros, consolidação ou portabilidade do código.

- **Modularidade**

É a propriedade de um sistema que foi decomposto em um conjunto de módulos (classes) coesos e acoplados fracamente. Tal propriedade provém do tradicional método do projeto estruturado [ PAG 88 ] de

INSTITUTO DE INFORMÁTICA
BIBLIOTECA

modularização de sistemas, utilizando-se das diretrizes de coesão e acoplamento entre módulos resultantes a fim de se obter um bom nível de reusabilidade e extensibilidade de software.

Através da definição da abstração e encapsulamento das classes, acrescentado das diretrizes de modularidade, existe a possibilidade se conectar às classes que formam um sistema. Uma das principais formas de relacionamento entre classes é a associação. **Associação** [COA 91] [RUM 91] descreve um conjunto de ligações que possuem estrutura e semântica comuns. Uma **ligação** é uma conexão física ou conceitual entre objetos. Uma associação descreve um conjunto de ligações em potencial do mesmo modo que uma classe descreve um conjunto de objetos em potencial.

- **Hierarquia**

Uma hierarquia é resultante de uma classificação ou ordenação de abstrações. As duas hierarquias mais importantes em um sistema complexo são a estrutura de classes ( hierarquia "é um tipo de" - que corresponde ao relacionamento de herança ) e a estrutura de objetos ( hierarquia "é parte de" - que corresponde ao relacionamento de agregação ).

**Herança** é o relacionamento através do qual atributos e métodos são compartilhados entre classes organizadas hierarquicamente. Uma classe pode ser definida grosseiramente e então refinada em sucessivas subclasses mais detalhadas. Cada subclasse incorpora, ou "herda", todas as propriedades da superclasse e adiciona suas próprias propriedades. As características de uma superclasse não precisam ser repetidas nas subclasses, deste modo a repetição de projeto e implementação pode ser reduzida. A reutilização provida pelo mecanismo de herança é uma das principais vantagens do paradigma de orientação a objetos. O termo **herança simples** indica que um subclasse possui apenas uma superclasse; **herança múltipla** ocorre quando uma subclasse possui mais de uma superclasse. A estrutura de herança entre classes é resultado do processo de generalização/especialização, onde uma superclasse representa abstrações generalizadas, e subclasses representam especializações nas quais características foram adicionadas ou modificadas.

**Agregação** define o relacionamento entre uma classe agregada que é composta por diversas classes componentes. As classes componentes "são parte da" classe agregada. Uma classe agregada é semanticamente um objeto estendido que é tratado como uma unidade em muitas operações, apesar de fisicamente possuir diversos objetos menores. Classes componentes podem ou não existir independentemente da classe agregada, bem como participar de outras agregações. O relacionamento de agregação é transitivo, isto é, se uma classe possui componentes, estes componentes podem possuir outros componentes. Agregação recursiva também é possível.

- **Tipo**

Tipo é a imposição de uma classe a um objeto, tal que objetos de diferentes tipos não podem ser intercambiados, ou no máximo, podem ser intercambiados de maneira muito restrita. Tipo não deve ser considerado como aspecto vital na orientação a objetos, uma vez que um sistema pode ser desenvolvido com linguagens fortemente tipadas, bem como com linguagens fracamente tipadas.

Entretanto a introdução de verificação de tipos pode trazer alguns benefícios [TES 81]:

a) Sem verificação de tipo, um programa na maioria das linguagens pode "parar" de formas misteriosas em tempo de execução;

b) Na maioria dos sistemas, o ciclo edição-compilação-depuração é tão tedioso que a detecção de erros o mais cedo possível é indispensável;

c) Declaração de tipos ajuda a documentar programas;

d) A maioria dos compiladores consegue gerar código-objeto mais eficiente se tipos são declarados.

A utilização de tipo também envolve os conceitos de polimorfismo e ligação dinâmica já apresentados.

Além dos conceitos já apresentados ainda existem outros que envolvem outras particularidades da orientação a objetos:

- **Relacionamento de utilização**

Tal tipo de relacionamento ocorre quando uma classe utiliza os serviços oferecidos (indicados na interface) por outra classe. Este relacionamento estabelece a ligação de cliente/servidor [MEY 88] entre classes e indica que os objetos podem trocar mensagens no sistema.

- **Classe postergada**

Uma classe postergada [MEY 88] [NER91], também chamada de abstrata [RUM 91], é uma classe que não possui instâncias diretas, mas que suas classes descendentes possuem. Classes postergadas organizam características comuns que podem ser herdadas pelos descendentes. Por outro lado podem definir apenas o protocolo de um método sem a implementação interna que deverá ser realizada pelos descendentes. Classes que possuem instâncias diretas não podem possuir métodos postergados e normalmente estão posicionados nas "folhas" de uma árvore que representa a organização de herança entre classes.

- **Classe genérica**

Uma classe genérica [MEY 88], também chamada de parametrizável [BOO 91] [RUM 91], é uma classe que serve de gabarito (formato) para outras classes. Este tipo de classe foi introduzido em linguagens de programação fortemente tipadas para permitir mais flexibilidade na definição de operações que podem ser aplicadas sobre várias classes distintas. Uma classe genérica deve ser instanciada com outra classe antes que instâncias possam ser criadas. A utilização de classes genéricas vem reforçar a reutilização de software uma vez que tais classes apresentam operações comuns sobre as diversas classes que lhe servem de parâmetro.

- **Relacionamento de instanciação**

Tal relacionamento surge quando da utilização de classes genéricas as quais necessitam de uma forma para expressar a sua instanciação para efetiva utilização. A partir deste relacionamento conecta-se a classe genérica com a classe que lhe instancia.

## 3.2 Metodologias de desenvolvimento de sofware

Uma metodologia para desenvolvimento de software pode ser entendida como um processo para a produção organizada de software, baseada em um conjunto de técnicas pré-definidas e notações convencionais. Normalmente uma metodologia apresenta uma série de etapas básicas que envolvem tarefas e notações associadas às mesmas. Tais etapas básicas usualmente são organizadas dentro do ciclo de vida de desenvolvimento de software, que envolve basicamente as etapas de análise, projeto, implementação e testes, e manutenção.

Estas etapas podem ainda ser divididas em diversas atividades que podem ser representadas no ciclo tradicional de um software chamado de "modelo cascata" (figura 3.1).

Figura 3.1: Modelo tradicional do ciclo de vida de software.

O número de atividades varia de autor para autor; entretanto, este modelo descreve o processo de desenvolvimento de software como uma série linear de atividades, cada qual devendo terminar antes que a próxima inicie, podendo apresentar eventuais retornos à tarefa anterior para revisão ou refinamento.

Em geral, o problema é inicialmente definido e os requerimentos dos usuários é analisado. É gerada uma especificação de requerimentos do usuário, normalmente utilizando-se termos usados no ambiente do problema em que o usuário executa suas funções. Também é gerada uma especificação de requerimentos de software onde são descritos, utilizando-se de linguagem do projetista de software, os requerimentos precisos do sistema. Até este ponto são executadas atividades relativas à etapa de análise que busca identificar respostas à questão **O QUE?**, na definição do problema. As tarefas seguintes envolvem o projeto do sistema, onde se busca identificar as respostas à questão **COMO?**. As tarefas de projeto preliminar para detalhado buscam progressivamente decompor um sistema em componentes mais detalhados que serão implementados, testados e finalmente liberados para utilização. A tarefa de manutenção busca corrigir e adequar o sistema para o usuário final.

O ciclo de vida de software descrito deste modo geralmente é executado baseado na interpretação do mundo real em termos de decomposição funcional. Decomposição funcional é apenas uma das possíveis formas de desenvolvimento de software baseadas em decomposição. A seguir são descritas algumas destas metodologias e introduzidas as características do desenvolvimento de software OO.

## 3.3 Metodologias baseadas em decomposição

- **Decomposição funcional**

Atualmente a grande maioria das metodologias de engenharia de software largamente utilizadas são baseadas no paradigma da decomposição funcional, que enfatiza a definição de procedimentos e algorítmos. Análise

estruturada [DEM 78] [GAN 79] [YOU 89] e projeto estruturado [PAG 88] são metodologias que se baseiam em decomposição funcional.

Por este tipo de decomposição, inicialmente o sistema é visto em alto nível em função de o que ele deve realizar. Durante as fases seguintes o sistema é detalhado indicando como são executados seus objetivos. Ferramentas de apoio a esta metodologia baseiam-se fortemente em fluxo de dados e incluem: diagramas de fluxo de dados, dicionário de dados (na análise) e diagramas de estrutura (no projeto).

A decomposição funcional é bem suportada por linguagens procedurais tradicionais sendo portanto um modo natural de expressar o projeto de sistemas. Apesar de algumas destas linguagens apresentar algum nível de modularização que permita o encapsulamento de dados, normalmente dados são armazenados em áreas de acesso global. Tal característica implica na ocorrência do "efeito cascata" onde a alteração em uma parte do sistema pode se propagar por outras partes, dificultando o trabalho de manutenção.

Além disso, um sistema considerado como realizando um único serviço (possuindo uma única função) dificilmente evolui para incluir novas estruturas de dados ou novas funções com alguma robustez.

- **Desenvolvimento estruturado de Jackson**

Na metodologia proposta por Jackson [JAC 83] não há uma distinção clara entre análise e projeto unindo-os numa mesma especificação estruturada. Inicialmente as técnicas propostas modelam o mundo real, impõem alguma cronologia e utiliza alguns conceitos de OO (métodos de estrutura de dados). Após a execução de passos fundamentais que devem ser executados seqüencialmente, o modelo resultante descreve o mundo real em termos de entidades, ações e ordem de ações.

A metodologia pode ser considerada como baseada também em decomposição funcional, entretanto, é dada maior atenção às estruturas de dados. Tal metodologia busca modelar sistemas considerando entidades (semelhantes a objetos), mas impõe uma ordem cronológica sobre estes

componentes. Para sistemas onde tal suposição é válida, a metodologia provê boas abordagens, especialmente porque ela fornece a mesma representação para estrutura de programas, sistemas e estrutura de dados que envolvem a seqüência, a iteração e seleção. Também é fornecida uma ferramenta para análise e interpretação gráfica que, por incluir as construções de iteração e seleção, possui uma abordagem diferente de diagramas de fluxo de dados.

- **Desenvolvimento OO**

Na decomposição de sistemas pela abordagem de OO, o sistema é visto como uma coleção de objetos. Durante o desenvolvimento OO um sistema é modelado em três modelos básicos: o **modelo estático**, o **modelo dinâmico** e o **modelo funcional**.

No modelo estático são definidas as classes de objetos participantes do sistema em termos de estrutura (atributos e métodos) e relacionamentos. A partir deste, os modelos dinâmico e funcional podem ser definidos.

O modelo dinâmico define os aspectos do sistema sobre tempo e seqüência das operações, em termos de estados de objetos, transições (mudanças) de estados e eventos que causam transições de estado nos objetos. O modelo dinâmico captura o aspecto de controle do sistema, que descreve a ordem e seqüência de operações, sem no entanto descrever o que as operações realizam ou como foram implementadas.

Aspectos relacionados à transformação de valores como funções, restrições e dependências funcionais são representados no modelo funcional do sistema.

Análise e projeto em alto nível são executados não somente em termos destes objetos mas também pelos relacionamentos existentes entre eles. O projeto detalhado, que inclui implementação de processos e especificação de estrutura de dados interna, é postergado até o mais tarde possível e é privativo a cada objeto, reforçando o conceito de encapsulamento. Desta forma, um sistema baseado na representação por objetos pode

permanecer mais flexível uma vez que mudanças na implementação podem ser mais facilmente realizadas sem implicar em mudanças no projeto.

Metodologias OO também são normalmente apoiadas por ferramentas gráficas para a especificação dos modelos representativos de um sistema.

Uma vez que muito do desenvolvimento de programação OO [REN 82] [STR88] é '*bottom-up*' a diferença entre projeto de programas e codificação é menos clara que em sistemas com implementação procedural.

Em contraste com análise estruturada de sistemas baseada na abordagem de decomposição funcional '*top-down*', a análise e projeto OO possuem muita relação com ambas as abordagens: '*top-down*' e, talvez mais predominantemente, '*bottom-up*'. Uma vez que um dos objetivos da implementação OO é a criação de classes genéricas que possam ser adicionadas a uma biblioteca de classes reutilizáveis, uma abordagem que considera tanto uma análise '*top-down*' quanto um projeto '*bottom-up*' simultaneamente, leva a definição de sistemas mais robustos.

## 3.4 Mudanças trazidas pela abordagem de OO

A utilização da abordagem de OO durante o desenvolvimento de software difere das abordagens tradicionais em vários aspectos. Tal diferença vem a provocar as seguintes mudanças na construção de sistemas:

- **Deslocamento do esforço de desenvolvimento para as fases iniciais do ciclo de vida**

A abordagem de OO move grande parte do esforço do desenvolvimento de software para as fases iniciais do ciclo de vida, principalmente para a análise. O esforço extra para emprego de mais tempo durante as fases de análise e projeto é compensado por uma implementação mais simples e rápida. Sendo o resultado do projeto mais claro e mais adaptável, futuras mudanças são mais fáceis de se executar.

- **Ênfase nos dados antes das funções**

Maior atenção é dirigida para dados ao invés das funções a serem executadas. Esta mudança de ênfase permite que o desenvolvimento de um sistema seja realizado sobre uma base mais estável e permite o uso de um único conceito de software durante todo o processo de desenvolvimento: o conceito de objeto. Todos os outros conceitos, como por exemplo, funções, relacionamentos, eventos, são organizados em torno dos objetos. Deste modo informações provenientes da análise não são perdidas nem transformadas quando o projeto ou a implementação são realizados.

A estrutura de dados (objetos) de uma aplicação e seus relacionamentos são mais imunes à mudança de requerimentos do que as operações sobre dados. Organizar um sistema ao redor de objetos ao invés de funções permite maior estabilidade no processo de desenvolvimento do que a permitida por abordagens de decomposição funcional. Encapsulamento com boa definição da interface que oculta detalhes da implementação interna são outras formas de se proteger contra efeitos de mudanças.

- **Processo de desenvolvimento não-disjunto**

Uma vez que pela abordagem de OO os objetos envolvidos no escopo do problema são definidos durante a fase de análise e estes são apenas estendidos e refinados durante o resto do desenvolvimento, sem mudança do universo de discurso (que sempre são os objetos), o processo de desenvolvimento é contínuo, e deste modo a mudança de fase no ciclo de vida é menos perceptível. Esta característica é melhor enfatizada se durante todo o processo de desenvolvimento forem utilizados modelos e ferramentas de apoio que permitam refinamento progressivo, ao invés de transformação ou conversão de modelos com ferramentas distintas.

- **Iteração ao invés de seqüência**

Apesar dos passos para o desenvolvimento de software OO serem apresentados em determinada ordem, o processo real é realizado de forma iterativa. Uma vez que o processo de desenvolvimento não é disjunto, isto facilita a repetição de passos dentro do detalhamento progressivo de um sistema. Cada iteração acrescenta e clareia características, ao invés de

modificar um trabalho já realizado, havendo menos chances de se introduzir erros e inconsistências.

## 3.5 Fases do desenvolvimento de software OO

No capítulo 5 serão apresentadas algumas metodologias de desenvolvimento de software OO existentes. Cada uma sugere sua própria forma para alcançar o efetivo desenvolvimento de sistemas.

Entretanto, entre metodologias OO pode-se notar que determinados passos básicos são seguidos para se alcançar certos objetivos. Segundo Henderson-Sellers [HEN 90] existem sete estágios básicos que metodologias para desenvolvimento de software OO possuem em comum:

1) *Desenvolvimento da especificação de requerimentos do sistema*: neste estágio são realizadas as tarefas de análise a alto nível em termos de objetos e seus serviços. Dependendo do esforço dispendido nesta etapa podem se originar especificações detalhadas como detalhes sobre tempo, utilização de hardware, estimativas de custos e outras documentações. [COA 91] e [SHL 88] propõem métodos específicos na análise orientada a objetos para este estágio.

2) *Identificação dos objetos*: tanto na fase de análise como no projeto lógico é necessária a identificação de objetos, seus atributos e métodos básicos. Objetos podem ser identificados a partir de elementos existentes no mundo real, bem como de entidades abstratas. Apesar da identificação de objetos (e, em última instância, de classes) ser inicialmente baseada na especificação de requerimentos do sistema é necessário já se antecipar o projeto de classes bem definidas que possam ser possivelmente reutilizadas em outros sistemas (aspecto abordado mais profundamente nos estágios 6 e 7). Como resultado deste estágio devem ser esboçados as possíveis classes de objetos participantes do sistema, bem como os seus atributos e métodos.

3) *Estabelecimento de relacionamento entre objetos*: neste estágio são estabelecidos os relacionamentos existentes entre objetos em

termos de serviços requeridos e serviços prestados. Para tal, neste momento é interessante alguma forma de notação diagramática comparável a um diagrama Entidade-Relacionamento [CHE 76] ou similar pela qual se possa especificar o relacionamento estático entre os objetos. Também poderiam ser utilizadas notações equivalentes aos tradicionais Diagramas de Fluxo de Dados [DEM 78] [GAN 79] para a modelagem da parte funcional do sistema, como proposto em [PAG 90] [RUM 91] [COA 91].

4) *Especificação de detalhes internos dos objetos*: à medida que o desenvolvimento do sistema começa a migrar da análise para o projeto é necessário iniciar o processo de especificação de detalhes internos dos objetos. Com base no produto resultante das etapas anteriores deve haver um refinamento do mesmo para se obter a especificação dos detalhes internos. Neste momento deve-se considerar os mecanismos envolvidos em desenvolvimento '*bottom-up*', buscando-se a utilização de componentes de projetos já definidos - um importante fator da estratégia de desenvolvimento de software orientado a objeto.

5) *Utilização de bibliotecas de classes*: como já observado no estágio anterior, é importante salientar a reutilização de definições, já existentes que estão contidas em bibliotecas de classes resultantes de sistemas já definidos ou de classes primitivas, através dos conceitos de cliente-servidor e contrato [MEY 88]. Neste ponto de desenvolvimento do sistema pode ser iniciada a implementação (codificação e testes) de classes de mais baixo nível.

6) *Introdução do relacionamento de herança*: como novos objetos vão sendo identificados dentro do projeto detalhado, a reavaliação do conjunto total de classes requerirá uma análise iterativa se novas superclasses ou subclasses que podem ser utilizadas posteriormente. Neste ponto existirá a necessidade de uma notação para se expressar o relacionamento de herança entre classes. Também neste estágio é necessária a criação de uma hierarquia bem definida a fim de que futuros projetos, que se baseiem no produto resultante do presente projeto, não necessitem reorganizar todo o esquema de heranças para suprir cada particularidade dos novos sistemas.

7) *Definição de agregação e/ou generalização de classes*: neste estágio são realizadas tarefas que visam a formulação de classes que sejam adequadas à reutilização em outros sistemas. Deve-se reavaliar classes definidas a fim de obter um sistema útil, robusto e confiável. Através do processo de generalização podem ser criadas classes mais genéricas que podem ser incluídas na biblioteca de classes pré-definidas. A curto prazo isto pode representar uma sobrecarga no tempo de desenvolvimento do sistema atual; entretanto, a longo prazo isto se transforma em uma redução no tempo de desenvolvimento de novos sistemas que podem ser beneficiados pela reutilização das classes definidas anteriormente.

Apesar da identificação destas etapas básicas, é importante ressaltar que o processo de desenvolvimento de software sob o paradigma de orientação a objetos não é baseado em estágios estanques, isto é, os passos apresentados podem, e geralmente o são, realizados em paralelo e mesmo em sobreposição, por vezes impossibilitando a caracterização clara da etapa em desenvolvimento. Tal característica permite indicar um modelo que representa melhor o ciclo de vida de desenvolvimento de software OO do que o modelo cascata: o "modelo fonte".

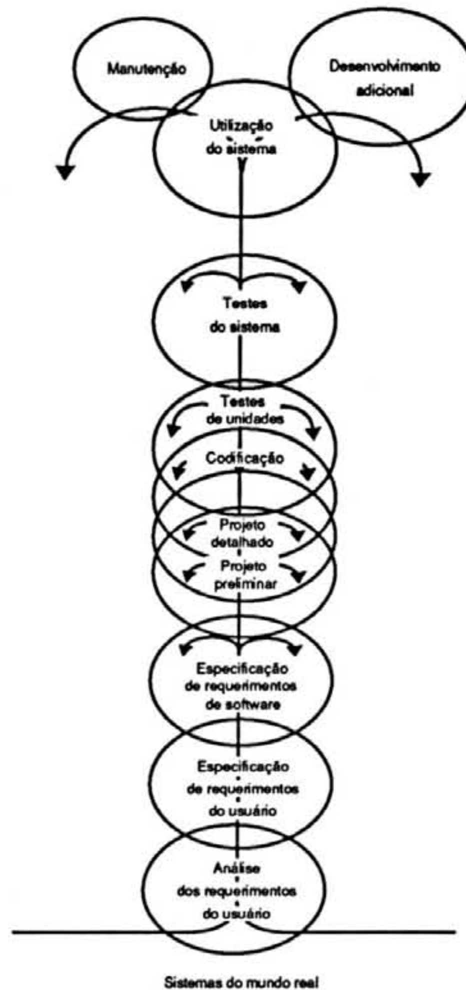



Figura 3.2: Modelo de um ciclo de vida de software OO [HEN 90].

O modelo fonte (figura 3.2), apresentado em [HEN 90], representa as diversas etapas do desenvolvimento de software OO, ressaltando o alto grau de iteração e sobreposição de tarefas. Bem como, representa o ciclo de vida de baixo para cima, onde a base representa os requerimentos provenientes do mundo real que servem de base para o resto do desenvolvimento do sistema, que culmina na utilização do sistema.

## 3.6 Necessidade de apoio diagramático

Com a ascenção do conceito de orientação a objetos, diversos autores propuseram técnicas, métodos e metodologias completas sob o novo paradigma, e atualmente coexistem diversas metodologias OO. Semelhante a

evolução do conceito de "estruturação" de software, inicialmente foram abordados aspectos relativos à programação, logo após o projeto e finalmente a análise, definindo metodologias completas.

Do mesmo modo que o desenvolvimento de software através de metodologias baseadas no paradigma de decomposição funcional, onde as tarefas são apoiadas por ferramentas gráficas, como DFDs, DEs, o desenvolvimento de software OO também necessita de apoio diagramático adequado para a especificação dos modelos representativos de um sistema.

Cada metodologia busca propor suas próprias notações. Aliás atualmente ainda não existe uma notação diagramática que possa ser chamada de "padrão". Deste modo, semelhante à diversidade de linguagens de programação OO e metodologias OO, existem atualmente diversas notações diagramáticas. Existe portanto a necessidade de se definir um conjunto mínimo de notações sobre as quais um sistema OO pudesse ser especificado. No capítulo seguinte é apresentado um conjunto de notações diagramáticas suficientes para a especificação de sistemas OO.

# 4 NOTAÇÕES DIAGRAMÁTICAS PROPOSTAS PARA APOIO AO DESENVOLVIMENTO DE SOFTWARE ORIENTADO A OBJETOS

É importante notar que desenvolver um sistema não é simplesmente desenhar diagramas, mas utilizá-los adequadamente como forma de auxílio durante este processo.

Um único sistema pode ser visualizado sob vários aspectos, gerando várias formas de interpretação e entendimento do mesmo. Para apoiar adequadamente as tarefas a serem executadas dentro de uma metodologia é necessário que os diversos aspectos relativos a um sistema possam ser representados por notações diagramáticas apropriadas.

A tentativa de se representar todos os aspectos num único diagrama pode causar, ao invés de objetividade e clareza, confusão a quem estiver desenvolvendo ou tentando entender o sistema.

Desta forma, é importante que a especificação do sistema não seja realizada sobre uma única notação diagramática, mas sim sobre a integração de notações diagramáticas, cada qual representando preferencialmente um aspecto distinto do sistema.

Um sistema orientado a objetos pode ser modelado sob três aspectos básicos: **estático, dinâmico** e o **funcional**. No modelo estático é identificada a estrutura das classes de objetos no sistema em termos de identidade, estrutura interna (atributos e métodos) e seus relacionamentos. O modelo estático serve de base para a modelagem dos outros aspectos do sistema. No modelo dinâmico identificam-se os aspectos do sistema relacionados ao tempo e a seqüência de operações, ou seja, o controle que existe dentro do sistema. No modelo funcional identificam-se os aspectos relacionados à transformação de valores em termos de funções, restrições e dependências funcionais.

Para a modelagem da parte estática do sistema são utilizados diagramas de classes e objetos. A modelagem dinâmica do sistema é obtida através de diagramas de estados e a modelagem funcional é baseada em diagramas de fluxo de dados e diagramas de ação. A seguir serão apresentadas as notações diagramáticas sugeridas para apoio à metodologia.

## 4.1 Diagrama de classes e objetos

Com base na notação apresentada nesta seção são representadas as características do modelo estático do sistema. Para tal a notação apresenta facilidades para que as características relativas à estrutura das classes de objetos do sistema (em termos de atributos e métodos), e à organização entre classes (em termos de relacionamentos) possam ser adequadamente representadas.

O diagrama de classes e objetos (DCO) resultante é uma rede contendo diversos elementos. Apesar de ser possível a representação de todos os elementos e suas ligações num único diagrama, a fim de evitar a confusão, normalmente os diversos aspectos da modelagem estática são representados separadamente. Isto implica que usualmente aspectos relacionados à estrutura de cada classe de objetos, os relacionamentos de associação, de agregação e de herança são representados, cada um, em diagramas separados.

A seguir são apresentadas as notações utilizadas na definição das características básicas de um diagrama de classes e objetos (DCO) na representação de classes e objetos, bem como os relacionamentos entre estes.

INSTITUTO DE INFORMÁTICA
BIBLIOTECA

### 4.1.1 Notações diagramáticas para representação de classes e objetos

Através desta notação se representa a estrutura de uma classe de objetos. Esta estrutura deve incluir informações sobre a identidade da classe (nome), atributos (dados), métodos (operações) e invariante (restrições sobre atributos e/ou métodos). Basicamente são representadas as características que fazem parte da abstração da classe, isto é, características que são importantes para outras classes do sistema.

Uma classe é representada (figura 4.1) por um retângulo que internamente possui o nome da classe e por um outro retângulo abaixo do primeiro que pode possuir três partes distintas: a primeira, formada por um retângulo interno que pode ser dividido em várias partes, onde são indicados os nomes dos atributos disponíveis na classe; a segunda, formada por um retângulo interno com cantos arredondados, que também pode ser dividido, onde são indicados os métodos; e a terceira parte onde é indicada a invariante da classe, que representa as restrições sobre os atributos e os métodos da mesma.

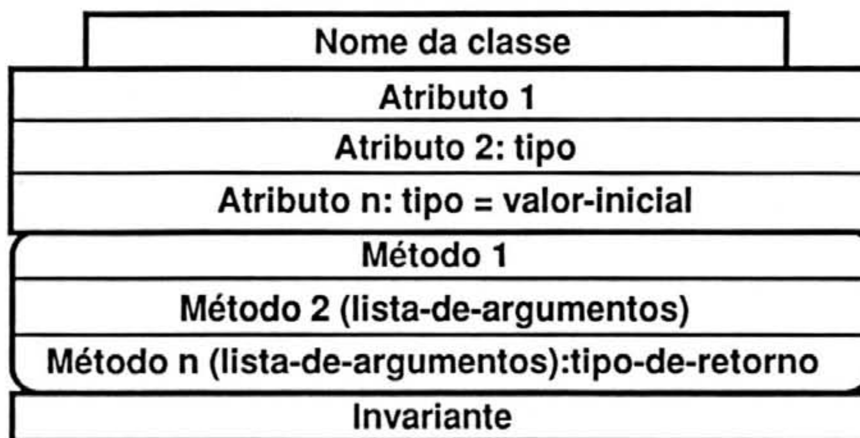



Figura 4.1: Notação básica para representação de uma classe.

Nota-se que o retângulo onde se indica o nome da classe é menor do que aquele que abriga o restante das características da classe para

se ressaltar a idéia de que as características ali representadas são exportáveis da classe, isto é, fazem parte da abstração da classe e podem ser "vistas" e utilizadas por outras classes do sistema.

A representação de atributos (retângulo simples) e métodos (retângulo com cantos arredondados) busca reforçar a idéia da distinção de conceitos entre os mesmos, como sugerido em [MAR 87], onde dados são representados por retângulos simples e operações por retângulos de cantos arredondados ou elipses.

Pode-se notar também que a indicação de atributos ou métodos na notação pode ser realizada de diferentes modos. Isto é importante para permitir que o modelo estático possa gradativamente ser refinado à medida que o sistema evolua da fase de análise para a fase de projeto e, conseqüentemente, que detalhes possam ser adicionados ao modelo. Tais detalhes também podem resultar do desenvolvimento dos outros modelos (dinâmico e funcional) do sistema.

Deste modo, um atributo pode ser simplesmente representado pelo seu nome de identificação, pelo nome de identificação acrescido da indicação do tipo de dado ao qual pertence ou, mais detalhadamente, pelo nome de identificação, tipo de dado e valor inicial. Um método pode ser representado apenas pelo seu nome identificador, ou mais detalhadamente pelo acréscimo de uma lista de parâmetros (com nome e tipo de cada parâmetro) para métodos que caracterizam procedimentos, ou então pelo acréscimo de uma lista de parâmetros e um tipo de retorno para métodos que caracterizam funções.

A representação da invariante da classe (que impõe restrições sobre os atributos e métodos da classe) pode ser realizada através de expressões lógicas simples.

Em determinadas ocasiões não há a necessidade de se visualizar todas as características da classe, bastando apenas tomar conhecimento do nome da mesma. A representação resumida de uma classe é realizada através

de um retângulo simples que possui internamente apenas o nome da classe, como pode ser visto na figura 4.2.

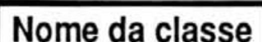



Figura 4.2: Notação para representação resumida de uma classe.

Este tipo de representação é interessante quando não há a necessidade de se visualizar a estrutura da classe (atributos, métodos e invariante), como por exemplo, na representação dos relacionamentos existentes entre classes, que dependendo da complexidade do sistema pode resultar numa rede complexa de linhas que pode ser agravada com a representação completa da estrutura da classe.

Entretanto também existem ocasiões em que o projetista ou proprietário da definição de uma classe tem interesse em visualizar todas as características desta classe, deste modo existe a possibilidade para representação estendida de uma classe (figura 4.3).

Atributos e métodos exportáveis são visualizados em modo realçado (retângulos maiores), enquanto que atributos e métodos internos são visualizados em modo simples (retângulos menores).

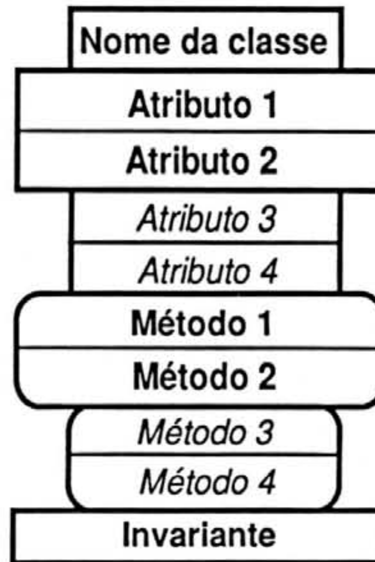



Figura 4.3: Notação para representação de uma classe completa.

Até o momento foram apresentadas as representações de classes de objetos. Entretanto, em determinadas fases do desenvolvimento do sistema é necessária a visualização de instâncias da classe, ou seja, dos objetos participantes do sistema. A representação de objetos pode ser utilizada para documentação de testes ou mesmo para exemplificação.

A representação de objetos é realizada pela instanciação de atributos das classes a que pertencem. Uma representação de classe pode possuir infinitas representações de objetos. Na figura 4.4 é apresentada a classe Funcionário e duas instâncias (objetos) da mesma, que são diferenciadas pelos valores de seus atributos.

Figura 4.4: Representação de objetos de uma dada classe.

### 4.1.2 Notações diagramáticas para representação de relacionamentos entre classes e objetos

Entre classes de um sistema podem existir diversos tipos de relacionamentos que necessitam ser representados adequadamente. Para a representação dos relacionamentos entre classes existem as seguintes notações:

a) **Associação:**

Este relacionamento é representado através de um vetor simples entre as classes envolvidas (figura 4.5). Apesar de certas notações utilizadas para a representação de associação serem apenas um segmento de reta indicando que a associação é bidirecional, optou-se pela utilização de um vetor, e conseqüentemente o significado de direção da associação, pois na prática a associação normalmente é implementada em apenas uma das classes. Caso a associação realmente interesse às duas classes envolvidas, a mesma poderá ser representada por um vetor bidirecional, e conseqüentemente existirão implementações para a ligação em ambas as classes.

Mesmo a representação da associação pode ser refinada de acordo com a fase de desenvolvimento do sistema. Numa abordagem inicial para a modelagem da associação entre duas classes, indica-se a mesma com a ligação de um vetor simples que una os retângulos que contêm os nomes das classes envolvidas. Posteriormente, quando do refinamento do modelo e especificação de características de implementação pode-se indicar na classe que implementa a associação qual o atributo utilizado para tal relacionamento, partindo dele o vetor da associação.

Na figura 4.5 é mostrada uma associação entre classes numa abordagem inicial. No refinamento do modelo poder-se-ia criar um novo atributo na classe de origem e indicar a partir deste atributo a associação entre classes.

As diversas possibilidades de cardinalidade de um relacionamento (de associação ou de agregação), utilizando-se a notação de vetor [MAR 87], são apresentadas na figura 4.6.

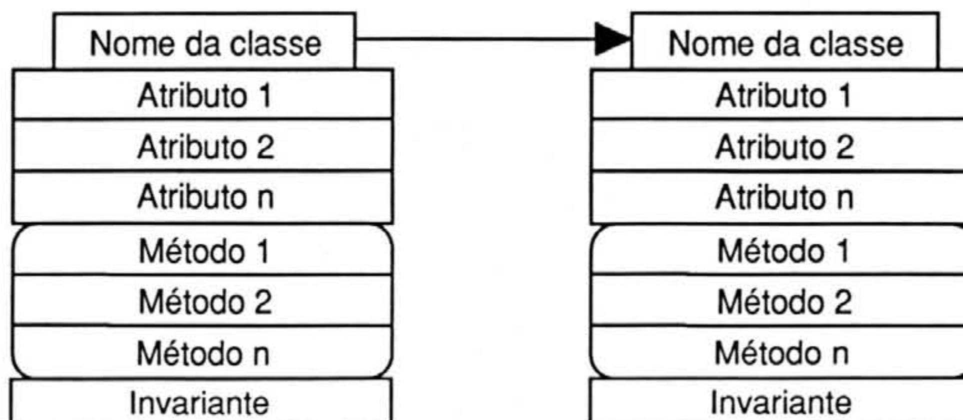



Figura 4.5: Representação do relacionamento de associação.

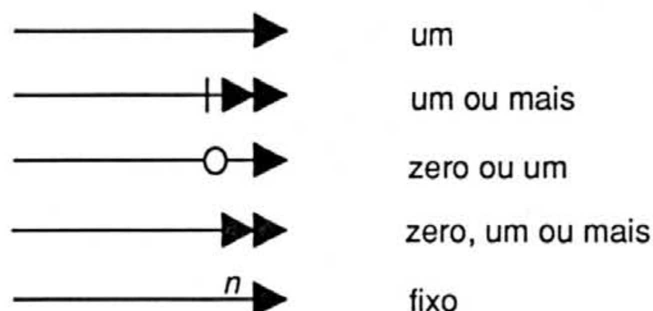



Figura 4.6: Cardinalidade de um relacionamento.

b) **Agregação**:

Pela definição deste relacionamento cria-se uma forte dependência entre as classes envolvidas. Estabelece-se o relacionamento de composição, onde uma classe (agregada) é composta por outras classes (componentes). A notação utilizada é um vetor que pode possuir cardinalidade, e que possui em sua base um pequeno losango (figura 4.7). Uma vez que o relacionamento de agregação não é simétrico é adequada a utilização de um vetor que indica qual a classe que agrega (origem do vetor) outras classes componentes (destino do vetor).

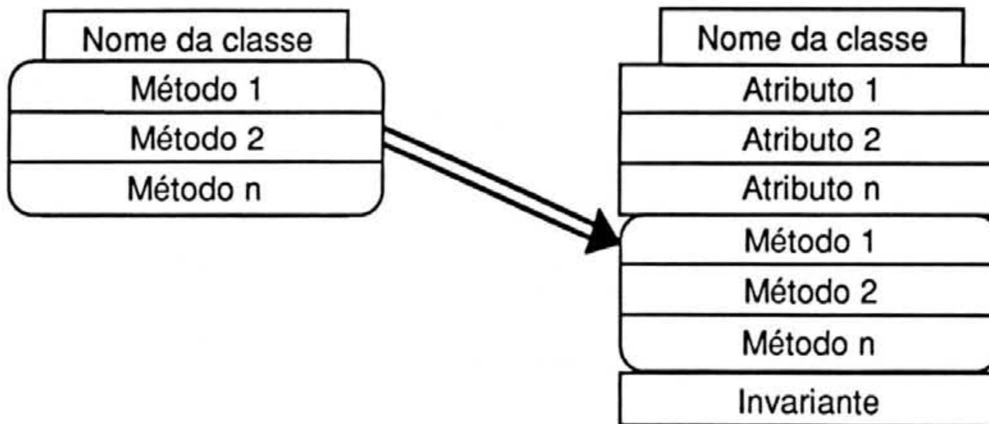



Figura 4.8: Representação do relacionamento de utilização.

d) **Herança**:

Através deste relacionamento estabelece-se a ligação entre uma classe (superclasse) e uma ou mais classes refinadas da primeira (subclasses). Tal relacionamento surge quando são utilizados os processos de generalização/especialização de classes dentro de um sistema.

A herança é representada através de um vetor largo preenchido (figura 4.9). A base do vetor indica a subclasse e o destino indica a superclasse. Como o relacionamento de herança também não é simétrico a utilização de vetores reforça a idéia de direcionamento do relacionamento.

Através do número de vetores originários de uma subclasse, pode-se definir o tipo de herança como simples (na figura, a classe inferior esquerda) ou múltipla (na figura, a classe inferior direita).

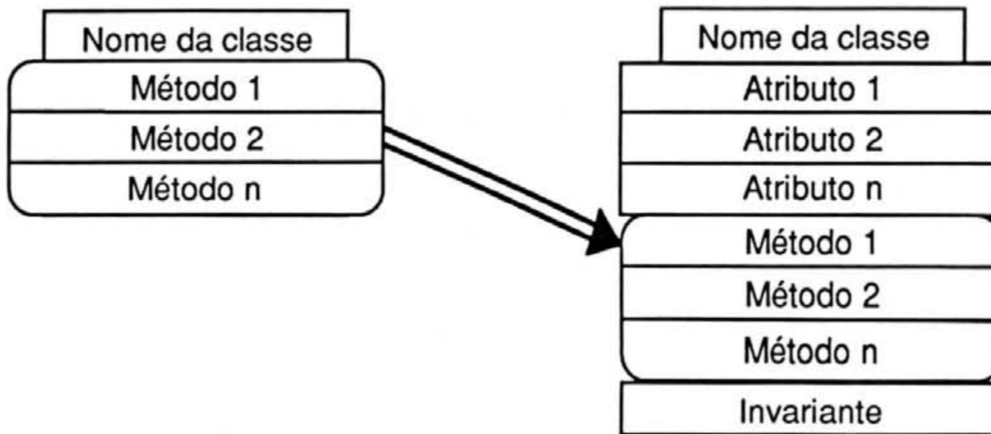



Figura 4.8: Representação do relacionamento de utilização.

d) **Herança**:

Através deste relacionamento estabelece-se a ligação entre uma classe (superclasse) e uma ou mais classes refinadas da primeira (subclasses). Tal relacionamento surge quando são utilizados os processos de generalização/especialização de classes dentro de um sistema.

A herança é representada através de um vetor largo preenchido (figura 4.9). A base do vetor indica a subclasse e o destino indica a superclasse. Como o relacionamento de herança também não é simétrico a utilização de vetores reforça a idéia de direcionamento do relacionamento.

Através do número de vetores originários de uma subclasse, pode-se definir o tipo de herança como simples (na figura, a classe inferior esquerda) ou múltipla (na figura, a classe inferior direita).

Como complemento à definição de classes postergadas, quando uma subclasse define completamente as características esquematizadas numa superclasse postergada, os métodos completamente definidos são marcados com um sinal de adição ( " + " ) do seu lado direito (figura 4.11).

Figura 4.11: Representação de características completamente definidas.

A utilização deste mecanismo também serve para indicar uma classe que possui atributos e/ou métodos que possuem novas implementações em relação à uma classe herdada.

Também buscando atender certas características particulares existe uma forma para a representação de classe genérica [MEY 88] ou classe parametrizável [BOO 91]. A indicação é realizada pelo acréscimo de um pequeno quadrado ( □ ) posicionado ao lado do nome da classe (figura 4.12).

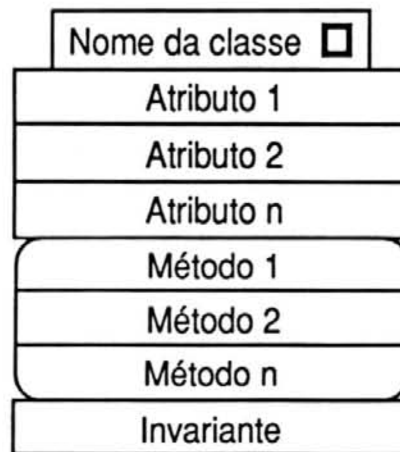



Figura 4.12: Representação de classe genérica ou parametrizável.

A instanciação de classes genéricas é realizada através de um sub-vetor (tracejado) que liga o vetor do relacionamento (de associação ou de agregação) entre classes, indicando qual é a classe utilizada para instanciar um classe genérica (figura 4.13).

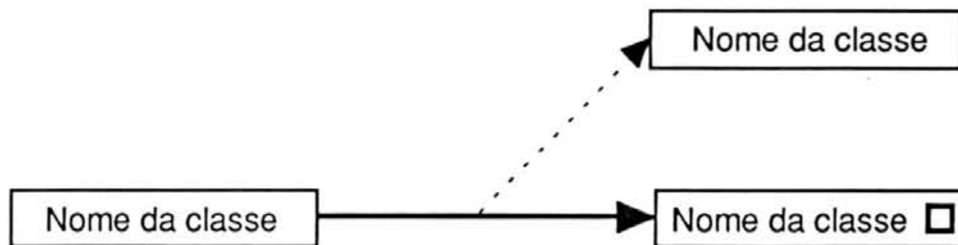



Figura 4.13: Instanciação de uma classe genérica.

## 4.2 Diagrama de estados

Até este ponto foram apresentadas formas para representação das características do modelo estático do sistema pela identificação da estrutura de classes de objetos e seus relacionamentos. No modelo dinâmico são identificadas as mudanças que ocorrem com os objetos e seus relacionamentos no decorrer do tempo. Neste modelo deve ser identificado o controle que existe no sistema e que seqüências de operações ocorrem em resposta a estímulos externos, sem considerações a respeito do que as operações fazem, nem mesmo como são implementadas.

Entretanto é através deste modelo que são identificados o fluxo de controle, seqüência e iterações de operações existentes no sistema. Para cada classe do sistema é utilizado um **diagrama de estados** (DE) que identifica o comportamento de seus objetos durante a execução do sistema. O conjunto de todos os DEs das classes componentes de um sistema se constitui no modelo dinâmico do mesmo.

O DE [HAR 88] é uma forma de representação gráfica de máquina de estados finitos e tem sido aplicado de diversas maneiras dependendo do contexto de sua utilização [MAR 87]. A utilização de DEs apresenta algumas vantagens sobre a utilização de diagramas de transição de estado, utilizadas em algumas metodologias de desenvolvimento de software orientado à objeto, como por exemplo a possibilidade de especificação de paralelismo de operações, chamadas a operações externas, condições complexas, organização hierárquica e modularidade.

No contexto de orientação a objetos, o DE busca identificar o comportamento de um objeto durante seu ciclo de vida [BEA 90]. Os **estados** em um DE representam os vários estágios que um objeto de uma determinada classe pode passar em relação aos valores de seus atributos e relacionamentos. As **transições** indicam os possíveis caminhos de mudança de um estado para outro. Para que um objeto passe de um estado para outro é necessário que exista uma transição unindo os dois estados e que um **evento** (estímulo externo) seja recebido pelo objeto, que pode se transformar em

operações no modelo estático. Associada a uma transição também pode ser definida uma ação que corresponde a funções no modelo funcional do sistema.

Um diagrama de estado básico é formado por nodos (estados) e por arcos (transições). Um estado é desenhado como um retângulo com cantos arredondados possuindo um nome opcional. Uma transição é representada por um vetor que liga nodos, sendo que este vetor possui opcionalmente o nome do evento associado.

Na figura 4.14, a seguir, pode-se observar um exemplo de DE. Através deste exemplo serão abordados alguns detalhes relativos a um DE.

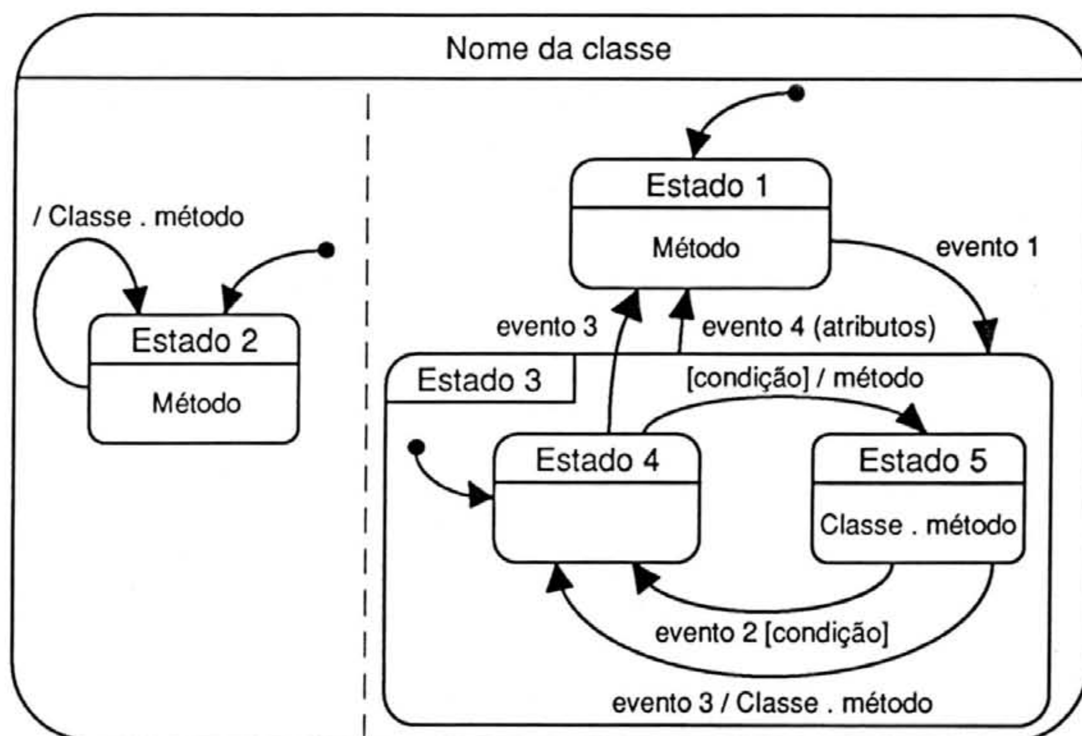



Figura 4.14: Exemplo de um DE.

### 4.2.1 Estados

Um estado no DE é uma abstração de valores de atributos e relacionamentos de um objeto. Cada estado pode possuir uma atividade associada (método do próprio objeto ou de um objeto relacionado) que é executada toda vez que o estado é alcançado no diagrama. A atividade associada a um estado pode possuir um tempo de execução fixo ou pode terminar caso um evento permita que se mude para outro estado. A representação da atividade quando envolve apenas métodos do próprio objeto é feita pela declaração do método invocado, em caso de método de objeto de outra classe, é indicada a classe e o método invocado (no exemplo: estado 5).

Apesar de um estado ser a abstração dos valores dos atributos de um objeto, nem todo estado é influenciado por todos os atributos; isto entretanto não indica que um determinado atributo seja inútil, pois o mesmo pode possuir importância em um dos outros modelos (estático ou funcional) do sistema.

Um estado é representado por um retângulo de cantos arredondados e o nome do estado no topo. Este nome é opcional pois em determinadas situações o nome do estado não é importante, mas sim a atividade associada. A declaração de atividades também é opcional e em caso de necessidade esta é especificada dentro do estado correspondente.

### 4.2.2 Transições

A passagem de um estado para outro é realizada caso haja uma transição entre os estados e é representada graficamente por um arco que liga dois estados distintos ou um estado a ele mesmo.

Associado a uma transição pode existir opcionalmente um evento e deste modo a transição só é realizada caso o evento ocorra. Um evento é um estímulo externo que o objeto recebe. Este evento pode ser um simples sinal, ou mesmo uma forma de comunicação de dados (valores de atributos)

entre objetos. Quando o evento se utiliza de atributos, estes devem ser especificados em uma lista, limitada por parênteses, colocada ao lado do nome do evento (no exemplo: evento 4). Um mesmo evento pode ocorrer mais de uma vez no DE (no exemplo: evento 3) indicando que o mesmo evento pode ser interpretado diferentemente dependendo do estado atual do objeto.

Caso haja a definição de uma condição (função lógica) - também opcional - associada a uma transição, a mudança de estado através desta transição só poderá ser efetuada caso a condição seja satisfeita. A representação é feita pela associação de uma condição (limitada por colchetes) a uma transição (no exemplo: transição entre os estados 4 e 5).

Eventos e condições podem ser combinados numa mesma transição (no exemplo: a transição entre os estados 5 e 4). Assim, a transição de estados só ocorre caso o evento especificado ocorra e a condição seja satisfeita.

Quando da transição entre estados, também pode ser opcionalmente especificada uma ação associada, representada após uma barra ("/"). Uma ação é a ativação de método do próprio objeto ou de outro relacionado. Do mesmo modo que atividades associadas a um estado, se a ação associada a uma transição envolve apenas um método do próprio objeto, este é apenas indicado junto à transição, caso contrário, é indicada a classe e o método invocado. A execução de uma ação pode ser condicional, caso sejam especificados um evento e/ou uma condição à transição; ou incondicional, caso nada seja especificado além da própria ação (no exemplo: ação associada ao estado 2). Neste último exemplo, sempre que a atividade associada ao estado termina, a ação associada à transição é executada, e quando esta termina, a atividade associada ao estado é novamente executada e assim por diante.

Ainda em relação a transição de estados, é importante notar que num DE, entre dois estados pode existir mais de uma transição (no exemplo: estados 4 e 5), porém as transições devem possuir eventos distintos. Também uma transição pode levar ao mesmo estado (no exemplo: estado 2), e caso haja alguma atividade associada ao estado, esta é novamente realizada. Uma

transição cuja origem é um círculo preenchido ( ● ) é a primeira a ser realizada quando do início do ciclo de vida do objeto. Uma transição cujo destino é um círculo preenchido dentro de um círculo simples ( ⦿ ) é a última a ser realizada no ciclo de vida do objeto.

### 4.2.3 Outras características

Com os elementos apresentados até aqui é possível a construção de DEs completos. Entretanto, os DEs apresentam outras facilidades relativas à sua organização.

De modo similar à estruturação de objetos no modelo estático, através de processos de especialização e agregação é possível se obter estruturas que facilitem a organização dos DEs. O processo de especialização permite que um estado presente em um diagrama possa ser expandido pela adição de detalhes, além de permitir que estados e eventos possam ser organizados em hierarquias de generalização possibilitando a herança de estrutura e comportamento comuns. O processo de agregação permite que um determinado estado possa ser decomposto em estados distintos com integração limitada entre eles. Tal processo permite a definição de concorrência entre estados, e estes podem indicar a concorrência entre objetos.

Uma atividade representada em um estado pode ser expandida em um subdiagrama com diversos estados. O subdiagrama resultante pode ser representado separadamente do diagrama inicial, e deste modo deve possuir pelo menos uma transição de saída. Os estados do subdiagrama gerado, neste caso, normalmente não são afetados por transições ocorridas no diagrama de mais alto nível.

Caso os subdiagramas resultantes sejam representados no mesmo diagrama, estes são definidos dentro do estado inicial (no exemplo, o estado 3 está expandido nos subestados 4 e 5), gerando um aninhamento de

DEs (no exemplo, o DE formado pelos estados 4 e 5 está aninhado em relação ao DE inicial).

Um DE aninhado representa uma hierarquia de estados. Os estados definidos no diagrama aninhado são todos refinamentos do estado no diagrama de mais alto nível, sendo que estes são afetados por transições ocorridas no nível mais alto (no exemplo: do subestado 4 ou 5 pode-se passar diretamente ao estado 1 caso o evento 4 ocorra), e podem interagir com outros estados (no exemplo: o subestado 4 interage diretamente com o estado 1 de nível mais alto).

Uma vez que um subestado interage diretamente com um estado de nível superior é necessário que todos os subestados sejam definidos no mesmo diagrama, caso contrário, é interessante se definir DEs separados a fim de reduzir a complexidade dos diagramas.

Caso um estado seja decomposto em diversos estados distintos, estabelece-se a concorrência de ações e atividades dentro de um objeto. A representação de concorrência entre estados é indicada pela divisão do diagrama de ação por uma linha tracejada (no exemplo, o estado 2 é concorrente aos estados 1 e 3 - conseqüentemente com o estado 4 ou 5).

Inerentemente o modelo dinâmico descreve a concorrência entre os objetos participantes do sistema que obedecem cada um o DE definido na classe a que pertencem. A definição do comportamento de um objeto agregado é o resultado da ligação entre todos os DEs de cada componente.

Do ponto de vista da herança, uma subclasse herda o DE da superclasse, porém os detalhes que fazem parte do refinamento/especialização da subclasse devem ser modelados em um diagrama independente/concorrente ao diagrama herdado.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

## 4.3 Diagrama de fluxo de dados

O **diagrama de fluxo de dados** (DFD) é utilizado para a definição do modelo funcional do sistema. Neste modelo deve ser identificado o que o sistema faz sem preocupação sobre quando são realizadas, apenas mapeando o significado das operações no modelo estático e ações no modelo dinâmico do sistema.

Através do modelo funcional deve ser identificado como os valores de saída em um processo são derivados de valores de entrada, bem como as restrições existentes entre estes valores. Para tal o modelo funcional é representado por um conjunto de DFDs através dos quais pode-se especificar o significado das operações e suas restrições.

DFDs são utilizados como ferramentas de especificação em diversas metodologias de desenvolvimento de software tradicionais baseadas no paradigma de decomposição funcional [DEM 78] [GAN 79] [YOU 89], entretanto, com poucas adaptações, podem ser utilizados para a especificação do modelo funcional de um sistema orientado a objetos.

No contexto da orientação a objetos, um DFD basicamente pode ser entendido como um grafo que mostra o fluxo de valores de dados de suas fontes (objetos) através de processos que os transformam em novos valores para seus destinos (em outros objetos). Em um DFD convencional não são mostradas informações de controle, como o momento de execução ou decisões sobre caminhos de execução de processos, que pertencem ao modelo dinâmico, apesar da existência de extensões a DFDs que incluem aspectos de controle (principalmente para aplicações de tempo real [WAR 86]). Em um DFD convencional também não é descrita a organização de valores nos objetos, que pertence ao modelo estático. Nas seções a seguir são apresentados vários aspectos relacionados a um DFD.

### 4.3.1 Processos

Um processo transforma valores de dados. Por exemplo, um processo pode executar operações aritméticas ou lógicas sobre dados para produzir algum resultado. O modelo funcional apresenta diversos processos que interagem entre si, porém neste modelo apenas são indicados os possíveis caminhos funcionais, não indicando se determinado caminho realmente ocorrerá. Através destes caminhos podem ser identificados diversos processos.

O sistema em si (visto do mais alto nível) pode ser considerado como um único processo. Por outro lado, processos de mais baixo nível (primitivas funcionais) representam as funções básicas do sistema e são implementados como métodos de um objeto. Pode-se notar portanto que os processos em um DFD podem ser expandidos em subprocessos (gerando novos DFDs em um nível inferior). Tal expansão de processos termina quando são definidos processos que realizam tarefas simples e bem definidas.

A representação diagramática de um processo pode ser um círculo ou elipse [DEM 78] [YOU 89] ou um retângulo de cantos arredondados [GAN 79] dividido internamente. A figura 4.15 apresenta os símbolos utilizados em DFDs segundo metodologias tradicionais de desenvolvimento de software.

A notação de um círculo ou elipse que internamente possui um nome que descreve a função realizada pelo processo é a mais tradicional. A notação proposta por Gane & Sarson para um processo é um retângulo de cantos arredondados e internamente é dividido em partes que definem um identificador (para controle de expansão de processos), a descrição do processo e opcionalmente o local físico onde o processo ocorre.

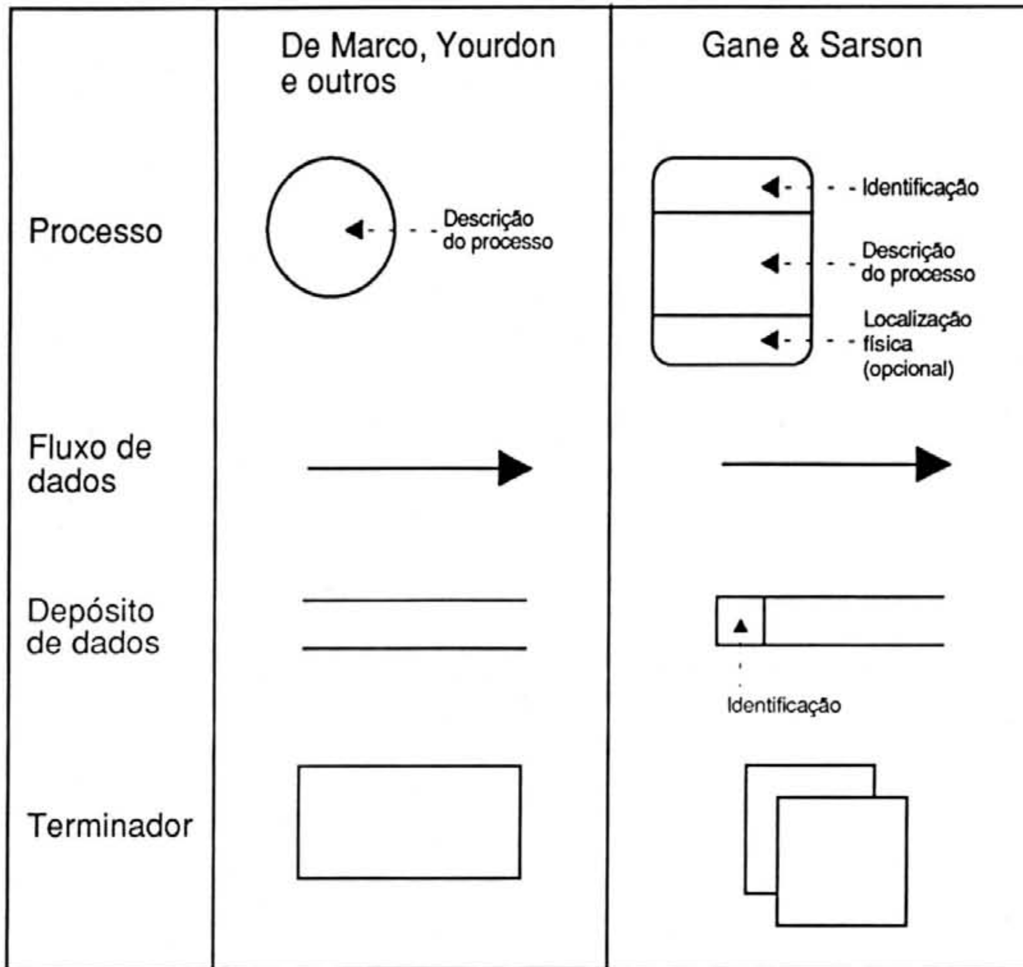



Figura 4.15: Símbolos utilizados em diagramas de fluxo de dados.

A utilização de um retângulo de cantos arredondados é mais adequada para o suporte computadorizado [MAR 87] pois permite que diversos fluxos possam ser conectados ao processo mais facilmente que uma representação de um círculo ou elipse. Internamente é indicada a descrição da função do processo (normalmente o nome do processo) e adicionalmente define-se um identificador numérico único para facilitar a integridade da operação de expansão de processos. A figura 4.16 apresenta a notação utilizada para representação de um processo.

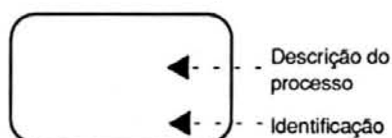



Figura 4.16: Representação de um processo em um DFD.

A identificação de um processo segue uma ordem progressiva a fim de que o controle de expansão de processos possa ser executado adequadamente. Por exemplo, um processo num nível superior que tenha como identificador 3, pode possuir diversos subprocessos no nível inferior cujos identificadores seriam 3.1, 3.2, etc.

Nos níveis mais baixos os processos representam funções básicas que podem ser implementadas como métodos de um objeto. O objeto destino, isto é, o objeto que efetivamente implementa um determinado método pode ser [ALA 88] [RUM 91]:

a) se um objeto é ao mesmo tempo a entrada e saída de um processo, então este objeto é o destino e os outros valores são os argumentos para o método;

b) se a saída de um processo é um depósito de dados, então o depósito de dados é o destino;

c) se a entrada de um processo é um depósito de dados, então o depósito de dados é o destino;

d) se a entrada ou saída é um terminador, então o terminador é o destino;

e) se a entrada é um objeto e a saída é parte deste mesmo objeto ou um "vizinho" deste objeto no modelo estático, então o objeto é o destino.

f) se nenhuma destas regras se aplica, então o objeto destino é implícito, não sendo nenhuma das entradas ou saídas, e freqüentemente o destino é descoberto pelo contexto de um subdiagrama completo.

### 4.3.2 Fluxos de dados

Fluxos de dados conectam a saída de um objeto ou processo a outro objeto ou processo, traçando o fluxo de dados através do sistema. O valor do dado não é alterado pelo fluxo de dados.

A representação de um fluxo de dados é um vetor (figura 4.15) entre o produtor e o consumidor do dado, sendo que este vetor é identificado pela descrição do dado, normalmente o próprio nome do dado.

Cada fluxo de dados representa um dado em algum ponto do processamento. Fluxos de dados internos a um diagrama representam valores intermediários dentro de um processamento e normalmente não possuem significado especial no mundo real. Por outro lado, fluxos de dados localizados nos limites de um diagrama indicam as entradas e saídas do sistema. Tais fluxos podem estar conectados a objetos. Os dados que identificam os fluxos podem ser valores simples representados por atributos no modelo estático, ou um objeto que implementa o processo ou que serve apenas de argumento à operação definida no processo.

Adicionalmente aos fluxos de dados, eventualmente um DFD pode apresentar também fluxos de controle. Um fluxo de controle representa um valor lógico que afeta o momento em que um processo pode ser executado, sendo que o fluxo de controle não leva efetivamente uma entrada de dados ao processo. Apesar de que os aspectos relacionados ao controle dentro do sistema pertencem primariamente ao modelo dinâmico, tais fluxos de controle podem garantir que determinadas dependências funcionais não sejam esquecidas. Um fluxo de controle é representado por um vetor simples porém tracejado, com o nome do valor lógico associado, e devem ser utilizados com parcimônia uma vez que se constituem em duplicação de informação com o modelo dinâmico.

### 4.3.3 Terminadores

Um terminador, também chamado de entidade externa, é um objeto ativo que participa externamente ao DFD produzindo e consumindo dados. Terminadores são ligados às entradas e saídas do DFD. Assim estão posicionados nos limites do diagrama, e deste modo eles são responsáveis por dados, ou como fontes ou sorvedouros dos mesmos.

As ações relativas aos terminadores não fazem parte da especificação do DFD, entretanto podem fazer parte da especificação do modelo dinâmico do sistema.

Existem basicamente duas representações gráficas para terminadores em um DFD, como pode ser visto na figura 4.15. Um terminador será representado segundo a notação de Gane & Sarson, onde são utilizados dois retângulos simples onde internamente é indicado o nome do objeto correspondente (figura 4.17). Fluxos de dados entre os terminadores e o diagrama representam as entradas e saídas do diagrama.

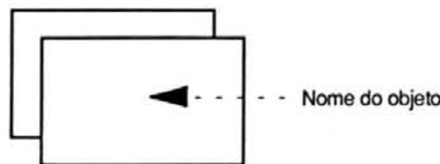



Figura 4.17: Representação de um terminador.

### 4.3.4 Depósito de dados

Um depósito de dados representa um objeto passivo dentro de um DFD que guarda dados para uso posterior. Ao contrário de um terminador, um depósito de dados não gera nenhuma operação por si próprio, apenas responde aos pedidos de armazenamento e acesso a dados.

A representação tradicional para um depósito de dados são dois segmentos de reta paralelos com a indicação do nome do depósito entre estes

(figura 4.15). Estendendo a notação proposta por Gane & Sarson um depósito de dados será representado por um retângulo simples (com o nome do objeto indicado internamente), a fim de facilitar as conexões (fluxos de dados) do DFD. A representação de um depósito de dados pode ser vista na figura 4.18.

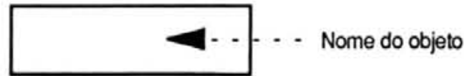



Figura 4.18: Representação de um depósito de dados.

Fluxos de dados entrando em um depósito indicam informações ou operações que alteram os dados do depósito, como por exemplo: inclusão de elementos, modificação de valores ou exclusão de elementos. Fluxos de dados saindo de um depósito indicam consulta a informações, que pode ser de um valor ou de um elemento completo. Em adição pode existir um fluxo de dados que entra e sai de um depósito (representado por um vetor bidirecional); tal fluxo indica que um dado é ao mesmo tempo entrada e saída de um depósito de dados. Poder-se-ia representar tal situação utilizando-se dois fluxos distintos, entretanto acesso e atualização de valores em um depósito de dados é uma operação comum, portanto sendo permitida tal representação.

A estrutura e as operações permitidas para o objeto que representa o depósito de dados devem ser adequadamente especificadas no modelo estático do sistema.

Tanto terminadores como depósitos de dados, bem como alguns fluxos de dados, podem ser entendidos como objetos. Em relação aos depósitos de dados, entretanto, existe uma diferença entre um fluxo de dados ser tratado como um valor único ou como um objeto. Isto é, um depósito de dados pode ser entendido como um objeto que agrupa muitos objetos de mesma estrutura. Quando um fluxo de dados entra ou sai de um depósito de dados, este fluxo pode representar um valor simples, que é armazenado na estrutura de algum objeto do depósito, ou então o fluxo representa um objeto completo que deve ser inserido ou retirado do depósito. Deste modo, um fluxo de dados que indica apenas um valor único possui a representação já apresentada, e um fluxo de dados que representa um objeto que deve ser

tratado pelo depósito de dados possui, ao invés da ponta simples, uma ponta vazia em sua extremidade (figura 4.19), como sugerido em [RUM 91].

Figura 4.19: Fluxo para tratamento de objeto em si.

### 4.3.5 Subdiagramas de fluxos de dados

Como já discutido quando da apresentação de processos, um DFD pode ser refinado em diversos subcomponentes, através da decomposição de seus processos. Um processo pode ser decomposto em um novo DFD num nível abaixo do DFD original, sendo que todas as entradas e saídas deste processo se constituirão nas entradas e saídas do novo diagrama, que pode possuir depósitos de dados que não aparecem no nível superior.

O conceito de níveis também é importante em relação à correspondência aos objetos que implementam os processos de um DFD, assim, um processo existente em um DFD em alto nível corresponde a uma operação em um objeto complexo, enquanto que um processo no nível mais baixo corresponde a uma operação em algum objeto básico que faz parte do objeto complexo.

Quando finalmente o refinamento de um DFD chega a um ponto em que os processos não podem mais ser refinados, pois descrevem apenas funções simples, estas funções devem ser especificadas como operações.

### 4.3.6 Especificação de operações

Em determinado ponto do desenvolvimento do DFD cada função executada pelos processos deve ser especificada como operações que possuem entrada, processamento e saída de dados.

Um processo em alto nível pode ser especificado em termos dos subprocessos resultantes do seu refinamento; entretanto, processos do nível mais baixo devem possuir uma definição mais especifica. A especificação das operações de tais processos pode ser realizada através de uma variedade de formas que incluem:

- pseudocódigo
- linguagem natural
- tabelas ou árvores de decisão
- equações matemáticas
- tabela de entradas e saídas de valores (enumeração)
- gramática de atributos
- especificação algébrica

Em uma primeira instância a especificação de uma operação poderia até mesmo ser realizada através de uma descrição em linguagem natural na qual relacionar-se-iam as entradas, as saídas, a lógica de transformação das entradas em saídas e as restrições impostas (tais restrições servem também para a definição de invariantes definidas no modelo estático). Porém à medida em que o desenvolvimento do sistema se aprofunde e detalhes devam ser adicionados, é interessante que haja alguma forma de apoio diagramático à descrição de operações em um DFD. Uma forma diagramática sugerida em [MAR 87] para especificação de operações seria a utilização de diagramas de ação.

## 4.4 Diagrama de ação

Como introduzido na seção anterior durante o processo de desenvolvimento do sistema haverá um determinado momento em que será necessária a especificação das operações existentes no sistema. Tais operações estão presentes no modelo funcional (DFDs) na forma de processos, bem como no modelo estático (DCOs) na forma de métodos.

A definição de tais operações pode ser feita através de vários modos. Entretanto, um apoio diagramático neste aspecto também seria importante. Como já discutido anteriormente a definição das operações depende da fase de desenvolvimento do sistema em execução, isto é, provavelmente numa abordagem inicial do sistema as operações não seriam definidas utilizando-se de algorítmos ou de construções da linguagem de implementação prevista. Portanto o apoio diagramático para este tipo de definição deveria apresentar a facilidade de refinamento progressivo, acompanhando o próprio processo de refinamento do sistema.

Notações diagramáticas tradicionais para o apoio à especificação de operações, como fluxogramas ou diagramas Nassi-Shneiderman [NAS 73], são extremamente detalhados. Seria necessária uma notação que permitisse que a especificação fosse gradativamente refinada. A notação diagramática que suporta tais características para este tipo de apoio é o **diagrama de ação** (DA) sugerido em [MAR 87].

Tal diagrama apresenta algumas vantagens em sua utilização como:

- permite que uma especificação possa ser refinada desde uma descrição em alto nível até detalhes de codificação;
- pode ser editada manualmente como através de uma ferramentas automatizadas;
- possui todas as construções básicas de linguagens de programação sem se basear em nenhuma específica;

- permite ligação com modelo de dados;

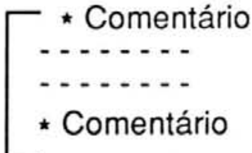
- permite a verificação de consistência em implementações computadorizadas.

Dentro da metodologia proposta a utilização de diagramas de ação se limitará à especificação de operações dos processos (no modelo funcional) e conseqüentemente dos métodos (no modelo estático). Na figura 4.20 a seguir é apresentado o sumário das notações utilizadas em um DA [MAR 87]:

**Colchetes**

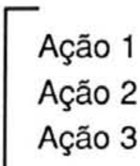
O colchete limita um conjunto de atividades que são executadas. Pode representar um programa, uma sub-rotina ou um bloco de código.

**Comentários**




Comentários (palavras, frases ou sentenças que não representam ação real) devem ser precedidos por um asterisco ( * ).

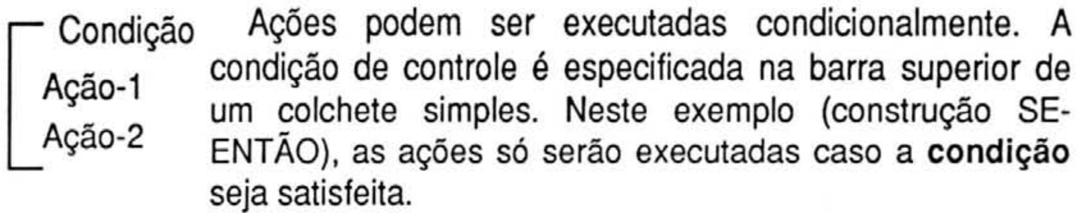
**Seqüência**



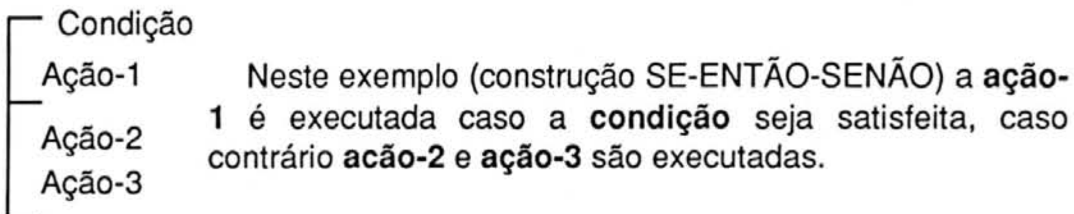

Uma ou mais ações podem ser incluídas dentro de um colchete. As ações são listadas uma após outra e são executadas na ordem em que estão listadas.

Figura 4.20: Sumário das notações de um DA.

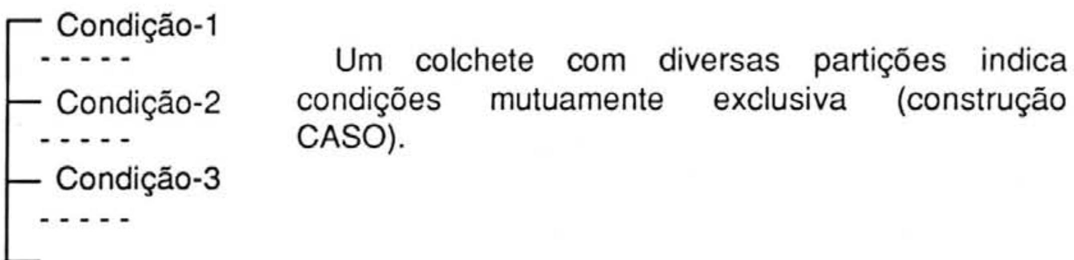
**Condição**




Ações podem ser executadas condicionalmente. A condição de controle é especificada na barra superior de um colchete simples. Neste exemplo (construção SE-ENTÃO), as ações só serão executadas caso a **condição** seja satisfeita.

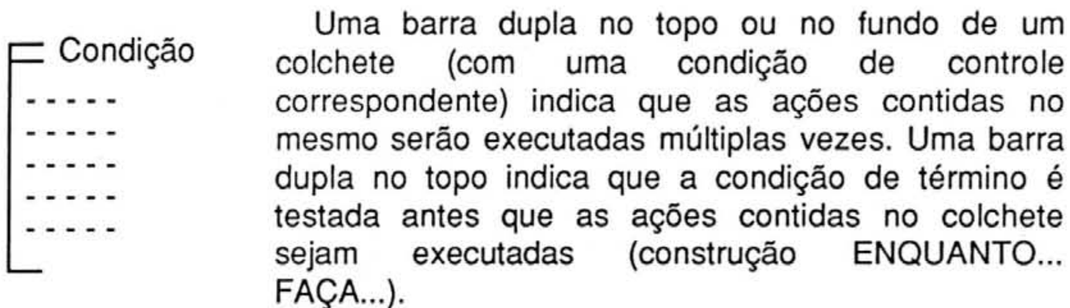



Neste exemplo (construção SE-ENTÃO-SENÃO) a **ação-1** é executada caso a **condição** seja satisfeita, caso contrário **acão-2** e **ação-3** são executadas.

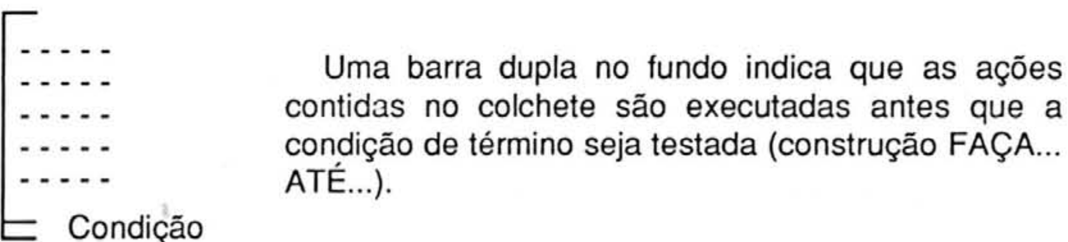



Um colchete com diversas partições indica condições mutuamente exclusiva (construção CASO).

**Repetição**



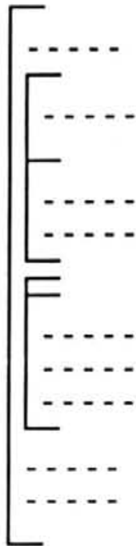

Uma barra dupla no topo ou no fundo de um colchete (com uma condição de controle correspondente) indica que as ações contidas no mesmo serão executadas múltiplas vezes. Uma barra dupla no topo indica que a condição de término é testada antes que as ações contidas no colchete sejam executadas (construção ENQUANTO... FAÇA...).




Uma barra dupla no fundo indica que as ações contidas no colchete são executadas antes que a condição de término seja testada (construção FAÇA... ATÉ...).

Figura 4.20: Sumário das notações de um DA. (continuação)

**Aninhamento**



Colchetes podem ser aninhados para indicar a estrutura hierárquica das ações.

**Formato retângulo**

O colchete pode ser estendido em um retângulo a fim de proporcionar a visualização das entradas e saídas de uma operação. O nome da operação é posicionada na barra superior do retângulo, as entradas no canto direito superior e as saídas no canto inferior direito.

Figura 4.20: Sumário das notações de um DA. (continuação)

Para cada operação existente deverá existir um DA para descrever sua função. Durante as fases preliminares do desenvolvimento a definição das operações poder-se-ia limitar a indicação de entradas e saídas e a indicação (através de comentários em linguagem natural) de sua função.

Entretanto, à medida que a especificação for refinada, maiores detalhes internos à operação devem ser adicionados, até o ponto em que seja possível gerar automaticamente um esquema de programa que implemente a lógica da operação.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

Os DAs devem ser utilizados para a especificação da lógica das operações em um DFD e conseqüentemente dos métodos dos objetos que implementam tais operações.

# 5 APOIO À METODOLOGIA DE DESENVOLVIMENTO DE SOFTWARE ORIENTADA A OBJETOS

Como já descrito no capítulo anterior, quando do desenvolvimento de um sistema orientado a objetos, este pode ser especificado através de três modelos distintos, porém complementares: o **modelo estático**, o **modelo dinâmico**, o **modelo funcional**.

Cabe considerar também que pela existência de diferentes tipos de sistemas, tais como[1] [RUM 91]:

- Transformação em lote - transformação de dados executada uma vez sobre um conjunto de entrada.

- Transformação contínua - transformação de dados executada continuamente à medida que as entradas se alteram.

- Interface interativa - um sistema dominado por interações externas.

- Simulação dinâmica - um sistema que simula objetos do mundo real que evoluem.

- Sistema de tempo real - um sistema dominado por restrições de tempo.

- Gerenciamento de transações - um sistema relacionado com armazenamento e atualização de dados, freqüentemente incluindo acessos concorrentes de diferentes localidades.

[1] A lista apresentada não é exaustiva, e também não indica que um determinado sistema possa ser classificado estritamente em uma categoria, normalmente podendo apresentar características de tipos diversos de sistema.

implica uma diferenciação no detalhamento dos modelos descritivos de um sistema, isto é, um sistema baseado em transformação em lote, por exemplo, possui um modelo funcional mais detalhado do que um sistema baseado em interface interativa, e este por sua vez possui um modelo dinâmico mais rico do que daquele.

Entretanto, uma metodologia de desenvolvimento de software deve propor passos, técnicas e apoio adequado para a especificação de todos os modelos, independente do tipo do sistema.

As notações diagramáticas apresentadas no capítulo anterior buscam apoiar adequadamente a especificação dos três modelos. Este apoio pode ser melhor definido dentro de um esquema de metodologia OO, onde são descritas tarefas, organizadas em etapas, que podem ser realizadas com a efetiva utilização de técnicas diagramáticas como apoio. Tal esquema busca satisfazer os requisitos básicos propostos em [HEN 90] durante a especificação dos três modelos básicos já apresentados e indicar a efetiva utilização das notações diagramáticas propostas.

## 5.1 Etapas básicas de uma metodologia de desenvolvimento de software orientada a objetos

### 5.1.1 Identificação de possíveis classes e objetos

Nesta etapa inicial através de estudos de uma especificação de requerimentos do sistema, de entrevistas ou de conhecimento próprio sobre o domínio do problema devem ser esboçados os objetos e as classes iniciais do sistema.

Objetos potenciais podem ser entidades físicas reais, bem como entidades conceituais. As classes de objetos, numa primeira abordagem, devem refletir o domínio do problema, sendo que construções de implementação, tais como listas encadeadas, sub-rotinas, devem ser evitadas;

exceto, é claro se tais construções são intimamente ligadas ao escopo do problema.

Neste momento também deve ser evitada a pré-identificação de relacionamentos entre classe, tais como herança, que pode precocemente forçar a estrutura das classes envolvidas.

Nomes de substantivos seguidos de adjetivos opcionais servem para a identificação das classes encontradas. Deve-se também procurar uma consistência no vocabulário utilizado para a nomenclatura das classes.

Após a identificação de todas as possíveis classes de objetos do domínio do problema, as classes que realmente importam devem ser extraídas através da eliminação de classes desnecessárias, tais como: classes redundantes (classes com nomes que possuem o mesmo significado), classes irrelevantes (classes com pouca ou nenhuma relação com o domínio do problema), atributos (nomes que são, na realidade, atributos pertencentes a uma outra classe), operações (nomes que são, na realidade, operações de outras classes), construções de implementação (quando não fazem realmente parte do escopo do problema).

Nesta etapa seriam esboçados os elementos básicos do modelo estático do sistema e, conseqüentemente, dos outros modelos. Para tal seria utilizada a notação para representação de classes e objetos que resultaria num DCO com classes distintas e separadas entre si e possivelmente com a identificação de alguns nomes de atributos e mesmo de métodos.

### 5.1.2 Identificação de relacionamentos entre classes

Esboçadas as classes pertinentes ao sistema, devem ser identificados os relacionamentos entre as mesmas, que podem ser definidos como associação, agregação, herança ou utilização.

Os relacionamentos normalmente podem ser identificados através de verbos na especificação de requerimentos do sistema. Qualquer

dependência entre duas classes se constitui em uma associação. Entretanto a distinção entre uma associação e uma agregação pode depender de conhecimento prévio sobre o problema; sendo a agregação uma associação com conotações extras, esta diferenciação pode ser postergada quando o esboço do sistema estiver mais claro.

Neste momento classes que compartilham mesmas características e comportamento podem ser organizadas através de herança entre si, bem como com classe pré-definidas, caso exista uma biblioteca de classes reutilizáveis. A utilização de herança para reforço do aspecto de reutilização de software deve ser levada em consideração.

Uma ligação entre duas classes que não se caracteriza em um relacionamento permanente, mas apenas pela utilização de um ou outro serviço de uma classe por outra se constitui no relacionamento de utilização.

Nesta etapa novamente o DCO deve ser utilizado, porém para permitir a ligação entre as classes definidas anteriormente. Através da utilização das notações apropriadas os relacionamentos de associação, agregação, hierarquia e utilização entre as classes são especificadas. Entretanto, proporcionalmente ao número de classes envolvidas no DCO a complexidade para o seu entendimento cresce. Deste modo seria interessante começar a especificar o DCO sob diversas visões, tendo por base os tipos de relacionamentos que pode haver entre as classes.

### 5.1.3 Identificação do "ciclo de vida" de um objeto

Até este ponto, são modeladas as características basicamente estáticas do sistema. Com base no modelo estático gerado é possível iniciar a especificação do "ciclo de vida" dos objetos do sistema. O "ciclo de vida" descreve as possíveis ações e atividades realizadas por um objeto durante a execução do sistema. Para tal especificação é definido um DE para cada classe de objetos existente no modelo estático do sistema.

Esta etapa visa indicar como um objeto se relaciona dinâmicamente com outros objetos, além de se formalizar os relacionamentos entre as classes envolvidas e identificar serviços prestados e solicitados que são necessários.

Através do DE podem ser identificados os métodos existentes na classe e mesmo atributos necessários a certas operações. O conjunto de todos os DEs se constitui no modelo dinâmico do sistema.

### 5.1.4 Definição interna de características das classes

A partir da definição constante no modelo estático e dinâmico, pode-se partir para definição do modelo funcional do sistema, onde são especificadas as operações que transformam dados dentro do sistema.

A especificação do modelo funcional é realizada através de DFDs que descrevem todo o sistema. Nesta etapa a utilização de DFDs em níveis é importante para se obter uma especificação gradual do sistema.

Inicialmente devem ser especificadas as entradas necessárias e as saídas produzidas pelo sistema, que pode ser modelado em um DFD de alto nível onde o sistema em si é representado por um único processo que interage com terminadores.

Com a definição do DFD inicial do sistema, este deve sofrer sucessivos refinamentos até que os processos presentes no nível mais baixo não possam ser mais refinados por representarem funções básicas. Neste momento então DAs são utilizados para a descrição de tais funções, e consecutivamente conectadas às classes de objetos adequadas.

No posterior refinamento dos métodos de uma classe são identificados os métodos internos à classe que não fazem parte da interface da mesma, mas que são necessários à funcionalidade da classe.

A definição de atributos nesta fase se resume à identificação de características específicas, como tipo e mesmo valor padrão ('*default*') que devem ser assumidos.

Neste estágio também podem ser especificadas regras de restrições (invariantes) sobre atributos e/ou métodos a fim de assegurar a utilidade da classe - um aspecto importante no relacionamento cliente-servidor. Em [MEY 88] a definição da invariante assegura um relacionamento cliente-servidor mais formal, chamado "contrato".

Aspectos relativos à implementação, como utilização de classes pré-definidas (biblioteca de classes oferecida por determinada linguagem de programação) e detalhalhamento dos DAs a nível algorítmico devem ser considerados.

### 5.1.5 Revisão de classes definidas

Neste estágio, que não necessariamente deve ser realizado como último, são realizados esforços com o objetivo de se obter a especificação de classes que apresentem características mais genéricas e úteis a futuros sistemas.

A partir do processo de generalização busca-se a especificação de classes genéricas e robustas o suficiente para que possam ser adicionadas a uma biblioteca de classes reutilizáveis. Tal processo deve ser realizado sobre o DCO do sistema com a criação e conseqüente relacionamento de novas classes.

Cabe ressaltar que o processo de desenvolvimento de software orientado a objetos não deve ser tomado apenas pela rígida e simples seqüência de passos sugeridos, mas sim pela constante busca de especificação adequada e cumprimento de objetivos básicos de cada etapa deste desenvolvimento. Uma metodologia de desenvolvimento de software orientada à objetos não pode ser tomada como sendo um processo monolítico nem unidirecional; a repetição na execução de tarefas é premissa comum a

qual se deve submeter quem se propõe a efetivamente desenvolver um sistema seguindo uma metodologia.

## 5.2 Comparação de apoio oferecido entre metodologias existentes

Atualmente coexistem diversas metodologias para o desenvolvimento de software orientado a objetos, bem como notações diagramáticas próprias para apoio a este desenvolvimento. Provavelmente uma diferença básica entre quaisquer metodologias propostas seja a ordem sugerida para a execução de tarefas específicas. Portanto aqui não serão abordados tais aspectos durante a comparação, que se resumirá a identificação de aspectos que envolvem o apoio à especificação dos modelos representativos de um sistema.

O conjunto de auxílios oferecidos por uma metodologia para a especificação de cada modelo será representado por um gráfico, como o sugerido na figura 5.1, no qual pode-se observar o apoio oferecido, em termos de notações diagramáticas e técnicas (que é representado por um círculo preenchido), aos modelos representativos de um sistema (que serão representados pelos eixos). Quanto mais distante da origem o círculo preenchido se encontra, maior o apoio oferecido pela metodologia para a especificação de determinado modelo.

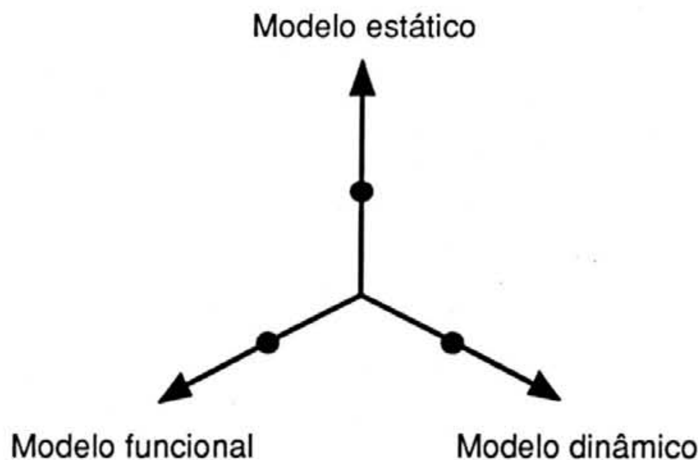



Figura 5.1: Gráfico de apoio oferecido aos modelos de um sistema.

A seguir são comparados os níveis de auxílio apresentados pelas metodologias já existentes.

### 5.2.1 Object Oriented Analysis (OOA)

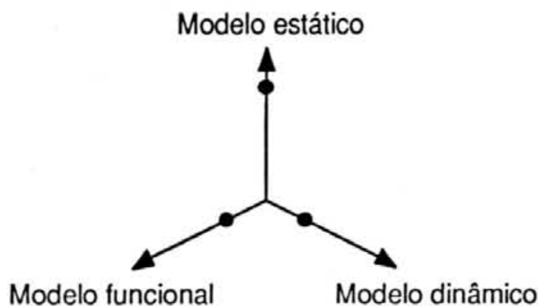



Figura 5.2: Apoio provido por OOA.

A metodologia proposta por Coad & Yourdon [COA 91] cobre basicamente a fase de análise, abrangendo tarefas relativas para esta etapa de desenvolvimento de sistemas.

Coad & Yourdon apresentam várias abordagens quanto à identificação das características de classes e objetos durante a especificação do modelo estático. Para tal são utilizadas notações diagramáticas para a representação de classes e objetos em termos de estrutura (atributos e métodos), bem como o inter-relacionamento entre classes e objetos. Adicionalmente são definidos gabaritos para a definição das características das classes. Entretanto não existe notação para expressar a existência de classes postergadas, que poderiam ser geradas durante o processo de refinamento da especificação do sistema.

Não existe muita ênfase na modelagem dinâmica do sistema que é apoiada por um diagrama de transição de estados simplificado, chamado de diagrama de estados de objeto, no qual não são especificados eventos, nem atividades que devem ser executadas. Do mesmo modo, a modelagem funcional basicamente é realizada pela identificação de comunicações, no modelo estático, entre objetos através de notação própria.

A especificação de operações (métodos) é realizada através da notação de diagrama de serviços, que se assemelha aos tradicionais

fluxogramas, com suas conseqüentes limitações. Invariantes sobre operações são expressas textualmente.

A metodologia OOA indica boas abordagens para a identificação de classes e objetos em termos de estrutura e relacionamentos, sendo interessante durante a fase de análise para a especificação do modelo estático do sistema.

### 5.2.2 Booch - Object Oriented Design (OOD)

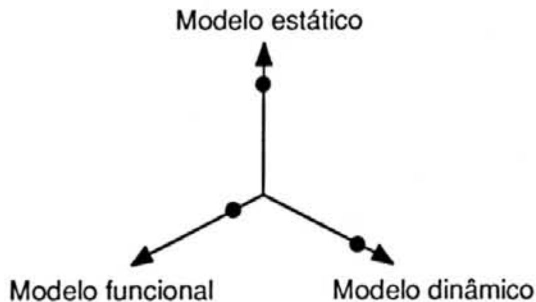



Figura 5.3: Apoio provido por OOD.

Booch apresenta em sua metodologia [BOO 91] tarefas relacionadas primariamente com o projeto de sistemas OO; indicando também algumas tarefas iniciais relacionadas à análise.

Para apoio a sua metodologia existem diversas notações diagramáticas para a representação de classes e objetos, suas estruturas e relacionamentos, bem como representação de classes em módulos (durante o projeto detalhado). Adicionalmente a metodologia reforça a utilização de gabaritos para a descrição textual de diversos aspectos do sistema.

A metodologia OOD possui grande embasamento do desenvolvimento do modelo estático e provê diversas notações diagramáticas para sua especificação. Entretanto não existe uma representação explícita para o relacionamento de agregação (que pode ser modelada como uma associação) nem para a definição de classes postergadas.

Quanto ao modelo dinâmico são utilizados complementarmente diagramas de transição de estados e diagramas de tempo, onde são identificados eventos e momentos que causam ações de objetos durante a execução do sistema. Para a modelagem dinâmica poderia ser utilizado um

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

diagrama de estado (como apresentado no capítulo anterior), ao invés de um diagrama de transição de estados, pois o diagrama de estado suporta a definição de características mais complexas.

Não existem passos sugeridos explicitamente, nem apoio diagramático adequado, para a modelagem funcional, que basicamente se restringe à identificação de operações pertencentes a cada classe de objetos e a especificação das características (textualmente) de cada operação em um gabarito à parte. A especificação de invariantes também é expressa textualmente nestes gabaritos.

Resumidamente, a metodologia proposta por Booch apresenta grande aprofundamento quanto à modelagem estática (tanto de classes quanto de objetos) e boa definição na modelagem dinâmica; entretanto, apresenta pouca ênfase na definição de dependências funcionais (modelo funcional) bem como a definição mais detalhada de operações existentes em um sistema.

## 5.2.3 Object Modeling Technique

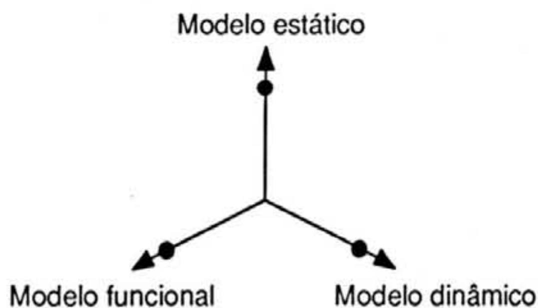



Figura 5.4: Apoio provido por OMT.

A metodologia proposta por Rumbaugh [RUM 91] é denominada OMT (*Object Modeling Technique*). Tal metodologia aborda as fases de análise e projeto OO e inclui abordagens quanto à definição dos modelos estático, dinâmico e funcional de um sistema.

A metodologia se baseia em diversas notações diagramáticas para apoiar a especificação dos modelos. Entretanto, no modelo estático, não existe definição para classes genéricas (parametrizáveis) que poderiam ser úteis na reutilização ou mesmo na definição de novas classes reutilizáveis.

A metodologia OMT provê apenas indicações quanto à definição de processos no projeto detalhado, não apresentando nenhuma notação

própria para apoiar tal tarefa. Restrições sobre métodos e atributos são esquematizadas durante a modelagem funcional e são apenas indicadas textualmente.

Basicamente, a metodologia OMT provê boas abordagens para a especificação dos modelos representativos de um sistema, durante as fases de análise e projeto. Entretanto, poderia haver uma notação adequada para a especificação da lógica interna de métodos durante o projeto detalhado.

### 5.2.4 Meyer - Object-Oriented Software Construction

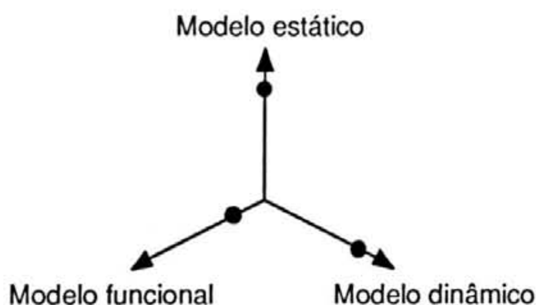



Figura 5.4: Apoio provido por Meyer-Nerson.

O trabalho apresentado por Meyer em [MEY 88] não se constitui realmente em uma metodologia, porém apresenta boas abordagens quanto à realização de um projeto OO, não se detendo muito a aspectos relacionados à análise nem à modelagem conceitual de sistemas.

O desenvolvimento de software apresentado no trabalho é altamente embasado em definições dependentes da linguagem Eiffel [INT 91] e são esboçadas algumas notações para representação de classes e objetos, em termos de suas estruturas (métodos e atributos), bem como de alguns relacionamentos.

Nerson apresenta em [NER 91] notações formalizadas para a representação de classes e objetos, e seus relacionamentos, denominada de BON (*Better Object Notation*), bem como tarefas relacionadas à análise OO. Tais tarefas enfatizam o desenvolvimento dos modelos estáticos e dinâmicos que são representados por notações próprias.

Existem indicações quanto ao desenvolvimento do modelo estático do sistema com notação apropriada. O modelo dinâmico é definido sobre o resultado do modelo estático com algumas alterações e basicamente se resume na definição de serviços prestados e requeridos para cada classe e o relacionamento de utilização, não indicando dependências temporais nem quanto a eventos.

A proposta de Nerson também é altamente baseada em construtores da linguagem Eiffel e busca reforçar a reutilização de software através da apresentação de uma biblioteca de classes pré-definidas. Entretanto não há grande preocupação quanto à modelagem funcional e conseqüentemente não apresenta apoio específico para tal.

O conjunto formado pelos passos sugeridos por Nerson e Meyer forma a base de uma metodologia que cobre as fases de análise e projeto OO. Entretanto haveria a necessidade de se especificar mais detalhadamente os passos sugeridos para o projeto de sistemas utilizando-se as notações sugeridas por Nerson.

## 5.2.5 Uniform Object Notation (UON)

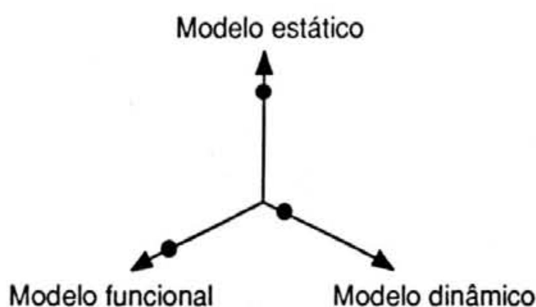



Figura 5.4: Apoio provido por UON.

UON, apresentada por Page-Jones em [PAG 90], também não representa realmente uma metodologia, mas sim uma proposta de uma notação diagramática que busca acomodar características referentes à classes de várias linguagens OO, bem como desenvolvimento de software baseado em objetos em linguagens estruturadas como COBOL ou FORTRAN.

Para a utilização da notação são sugeridos alguns passos para o desenvolvimento de software OO. UON provê o desenvolvimento do modelo estático utilizando-se o <u>diagrama de definição de classes</u> em conjunto com o

diagrama de herança. Entretanto não há definição explícita para a diferenciação dos relacionamentos de associação, agregação ou utilização, nem quanto à cardinalidade de tais relacionamentos. Devido à tentativa de capturar apenas características comuns a várias linguagens de programação a notação sugerida não suporta a definição de classes genéricas nem postergadas.

A notação provê também um diagrama de comunicação de objetos através do qual pode-se especificar o modelo funcional do sistema em termos de serviços prestados e requeridos, bem como a passagem de mensagens (dados) entre objetos. Em adição, também existe o diagrama de métodos internos onde pode ser especificada a interação entre métodos das própria classe. Porém, não há suporte para a definição mais detalhada de tais métodos, nem para a especificação de invariantes das classes.

Não existe suporte específico para a definição do modelo dinâmico, portanto aspectos relacionados a controle no sistema ficam embutidos apenas nos modelos estático e funcional.

A notação UON apresenta ainda algumas particularidades como suporte à função *friend* da linguagem C++ [STR 86]; bem como a especificação de relacionamentos "indeterminados" entre classes através do diagrama de cooperação de objetos.

Em resumo, UON busca ser uma notação básica para desenvolvimento de software sob qualquer metodologia. Entretanto, esta "padronização" limita a própria notação quanto à captura de certos detalhes envolvidos no desenvolvimento de software OO.

Como pôde ser observado, as várias metodologias de desenvolvimento de software orientado a objetos possuem diferenças quanto ao apoio prestado durante a especificação dos modelos representativos de um sistema.

As notações diagramáticas apresentadas neste trabalho buscam satisfazer e cobrir os aspectos abordados por estas metodologias, bem como servir de complemento às notações diagramáticas particulares de cada uma.

# 6 AMBIENTE DE APOIO

Este trabalho abordou até o momento diversos aspectos relacionados à utilização de notações diagramáticas para a especificação de sistemas sob metodologias de desenvolvimento de software orientado a objetos.

Entretanto, a especificação de um sistema através de notações diagramáticas realizadas de modo manual é uma tarefa árdua e desencorajante, pois a dificuldade de se gerenciar uma especificação gráfica, sem um auxílio automatizado, é proporcional à complexidade do sistema em desenvolvimento.

Deste modo, neste capítulo são esboçadas as facilidades básicas necessárias de ambiente para a efetiva utilização de técnicas diagramáticas.

## 6.1 Editores diagramáticos

O principal elemento necessário para apoiar as técnicas diagramáticas sugeridas neste trabalho é logicamente um editor diagramático. Um editor diagramático é uma ferramenta de software que através de uma interface gráfica permite que diagramas possam ser facilmente criados e alterados, relacionando e identificando os componentes do sistema a desenvolver. Diferente de um editor gráfico comum, um editor diagramático possui um conjunto pré-definido de elementos básicos da notação diagramática que suporta e regras de ligação entre estes elementos.

Deste modo, um usuário de um editor diagramático que suporta diagramas Entidade-Relacionamento [CHE 76], por exemplo, só poderá editar diagramas com os elementos básicos permitidos para este tipo de diagrama, com regras específicas para a sua edição. Entretanto, o trabalho de se desenhar, alterar ou eliminar um elemento, bem como realizar verificações sobre a correção do diagrama em edição fica ao encargo do editor diagramático, liberando o usuário para tarefas mais significativas.

Além disso, os editores permitem que os diagramas editados possam ser armazenados e recuperados facilmente, permitindo que módulos de um sistema possam ser reutilizados em outro sistema, reforçando desta maneira, mesmo à nível diagramático, a reutilização de software.

Atualmente existem diversos editores específicos para notações diagramáticas distintas. No contexto do projeto **AMADEUS** (Ambientes e Metodologias Adaptáveis de DEsenvolvimento Unificado de Software), em desenvolvimento no Curso de Pós-Graduação em Ciência da Computação da Universidade Federal do Rio Grande do Sul, no qual o presente trabalho se insere, existe implementado um Editor Diagramático Generalizado [MEL 89] que permite a criação de editores específicos para notações diagramáticas desejadas.

Adicionalmente a este editor existem trabalhos correlatos que buscam enriquecê-lo: [SIL 89] descreve a possibilidade de formatação automática de diagramas e [YAM 91] descreve a geração de gramática de atributos como forma de integração com editores textuais dirigidos por sintaxe [PRI 84] [ESP 89].

Tal editor permitiria portanto a criação de editores específicos para as notações diagramáticas propostas: DCO, DE, DFD, DA, durante o processo de desenvolvimento de software.

## 6.2 Editores textuais/diagramáticos

Além de um editor diagramático existe definido um editor dirigido por sintaxe [FAV 89] para suporte a diagramas utilizando-se do mecanismo de gramática de atributos para controle das características da notação diagramática especificada.

Tal editor permitiria a edição de diagramas de determinada notação com a verificação automática da correção. A utilização de tal editor

seria interessante para a especificação de DAs, nas quais poder-se-ia verificar automaticamente a correção da especificação sendo editada.

## 6.3 Dicionário de dados

Além de editores diagramáticos específicos é importante que haja um mecanismo que integre tais ferramentas a fim de compartilhar informações entre as mesmas. Este mecanismo seria provido com a existência de um dicionário de dados. Um dicionário de dados é uma ferramenta que, no contexto apresentado, permitiria que informações comuns aos editores diagramáticos fossem compartilhadas e a integridade das mesmas fosse verificada constantemente. Em [MRA 89] é descrita a implementação de um dicionário de dados para a integração de editores diagramáticos específicos.

## 6.4 Editor diagramático para apoio às notações propostas

Apesar da existência de diversas facilidades disponíveis, infelizmente estas não se adequam completamente à utilização necessária neste trabalho, pois:

- o editor diagramático generalizado não permite a verificação efetiva de semântica de elementos de uma notação diagramática;

- o dicionário de dados não permite que ferramentas distintas ( no caso, diversos editores diagramáticos ) sejam integradas;

- efetivamente ainda não existe um editor diagramático dirigido por sintaxe;

- a implementação das ferramentas citadas está baseada em equipamentos diversos, o que dificulta a integração;

- existe um esforço de padronização quanto ao equipamento/linguagem a ser utilizado para a definição de novas ferramentas no projeto AMADEUS;

- as ferramentas existentes não suportam as particularidades do paradigma de orientação a objetos.

Deste modo, a partir do conhecimento pré-adquirido durante o desenvolvimento das ferramentas já citadas será definido e implementado um editor diagramático específico para as notações diagramáticas propostas baseado em uma linguagem orientada a objetos.

A especificação de tal editor será apresentada através dos seguintes anexos:

- Anexo 1: Especificação dos requisitos do sistema.

- Anexo 2: Diagramas de Classes e Objetos para a definição do modelo estático do sistema (análise).

- Anexo 3: Diagramas de Classes e Objetos para a definição do modelo estático do sistema (projeto).

- Anexo 4: Diagramas de Estado para a definição do modelo dinâmico do sistema.

- Anexo 5: Diagramas de Fluxo de Dados e Diagramas de Ação para a definição do modelo funcional do sistema.

# 7 CONCLUSÃO

O presente trabalho buscou identificar técnicas diagramáticas adequadas que efetivamente sirvam de apoio ao desenvolvimento de software orientado a objetos.

Muitas metodologias sugerem a utilização de notações diagramáticas durante o desenvolvimento de sistemas, pois diagramas:

- auxiliam na clareza do raciocínio
- servem de meio de comunicação entre membros de equipes de desenvolvimento
- auxiliam na manutenção
- auxiliam na depuração
- servem como documentação

Com a introdução do paradigma de orientação a objetos no desenvolvimento de software, novos conceitos foram introduzidos, surgindo a necessidade de um apoio diagramático adequado para apoiar as tarefas específicas propostas pelas metodologias baseadas neste paradigma.

Sugeriu-se então um conjunto de técnicas diagramáticas básicas para a especificação dos três modelos representativos de um sistema: o estático, o dinâmico e o funcional.

Com base nas técnicas diagramáticas sugeridas o desenvolvimento de software orientado a objetos sob um esquema básico de tarefas, ou mesmo sob metodologias existentes, pode ser adequadamente executado.

Entretanto, para a efetiva utilização das notações propostas é necessária a existência de alguns recursos e facilidades básicas. Neste caso, um editor diagramático específico que suporte as notações sugeridas é definido utilizando-se as próprias notações para a especificação de sua implementação.

## 7.1 Trabalhos futuros

Para a efetiva utilização das técnicas sugeridas neste trabalho seriam importantes:

- a definição de um dicionário de dados orientado a objetos para armazenamento mais eficiente dos vários modelos representativos de um sistema, como também, para permitir a integração de outras técnicas diagramáticas, ou de integração com outras metodologias.

- a agregação de editores diagramáticos/textuais dirigidos por sintaxe que permitam a edição de diagramas com maior controle semântico, bem como, para facilitar a criação automática de código.

- a unificação com um gerente de hipertexto para permitir que diagramas possam ser compartilhados entre vários componentes de uma equipe de desenvolvimento.

# ANEXO 1 Especificação dos requisitos do sistema.

**Escopo do problema:**

Desenvolvimento de um ambiente gráfico para o desenvolvimento de software orientado a objetos que integre editores diagramáticos específicos às notações sugeridas no presente trabalho para apoio à modelagem estática, dinâmica e funcional de sistemas.

**Necessidades**

- Desenvolvimento de editores diagramáticos específicos para suporte a:

  - Diagramas Classe/Objeto

  - Diagramas de Estados

  - Diagramas de Fluxo de Dados

  - Diagramas de Ação

- Integração dos editores diagramáticos específicos.

- Interface comum aos editores.

- Possível integração com outros editores diagramáticos específicos.

## Contexto da aplicação

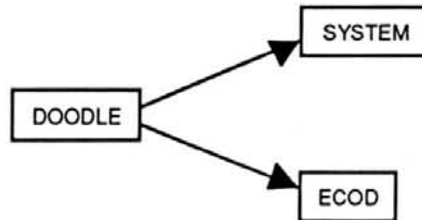
Para a utilização do sistema o usuário contará com uma estação de trabalho gráfica, um 'mouse' e um teclado. A estação de trabalho permite o suporte de elementos gráficos como pontos, linhas, janelas, etc. O 'mouse' será utilizado para a maior parte da comunicação do usuário com o ambiente. O teclado será utilizado para a entrada de informações textuais.

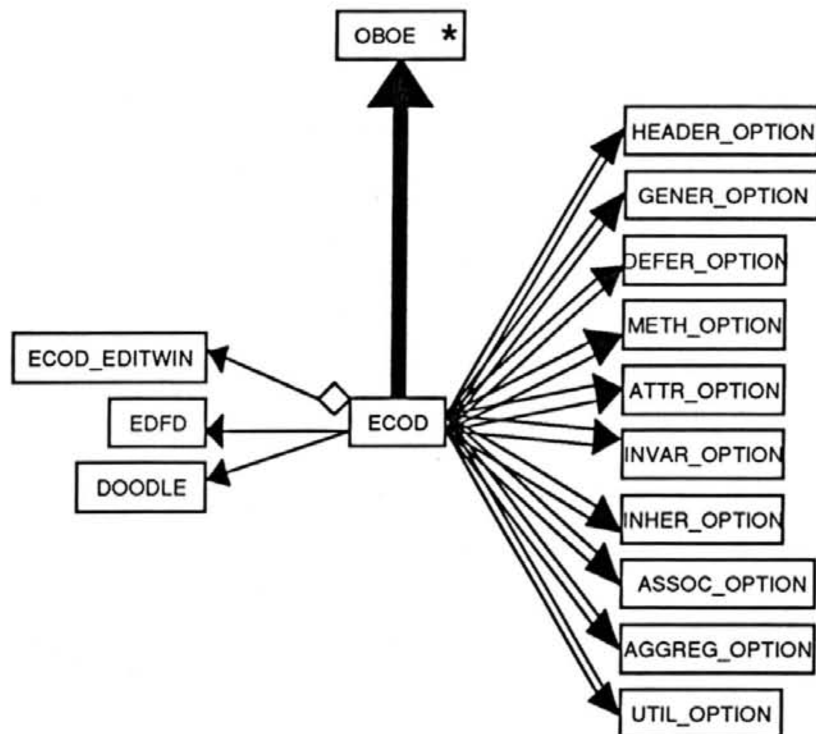
Um editor diagramático basicamente apresentará facilidades para a construção dos diagramas da técnica diagramática específica com a apresentação de menu de opções das primitivas gráficas bem como operações válidas sobre as mesmas. Deve-se buscar um padrão na interface dos editores gráficos, sendo que os editores diagramáticos específicos serão integrados a partir do editor de classes/objetos.

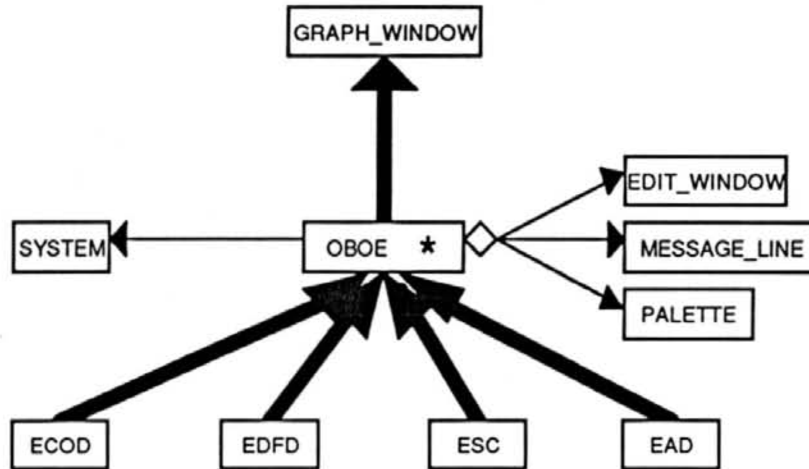
Para a implementação utilizar-se-á a linguagem Eiffel [ INT 91 ], bem como as classes pré-definidas pertencentes à biblioteca oferecida por esta linguagem.

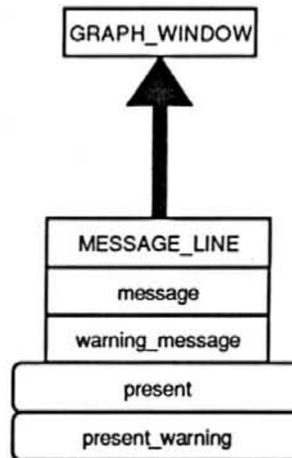
Maiores detalhes sobre as operações e características específicas de cada notação diagramática estão inseridos no texto deste trabalho.

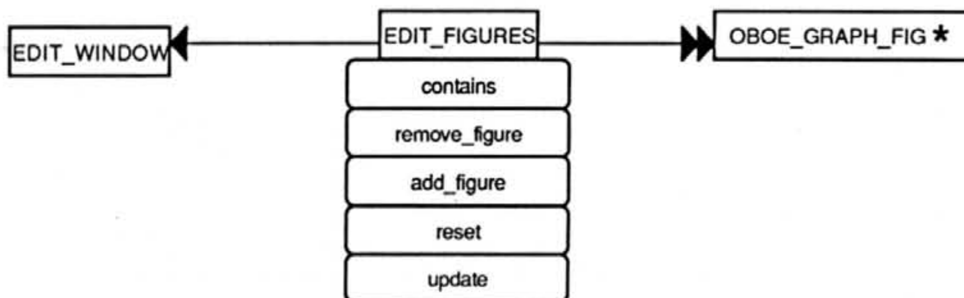
# ANEXO 2 Diagramas de classes e objetos para definição do modelo estático do sistema (Análise)

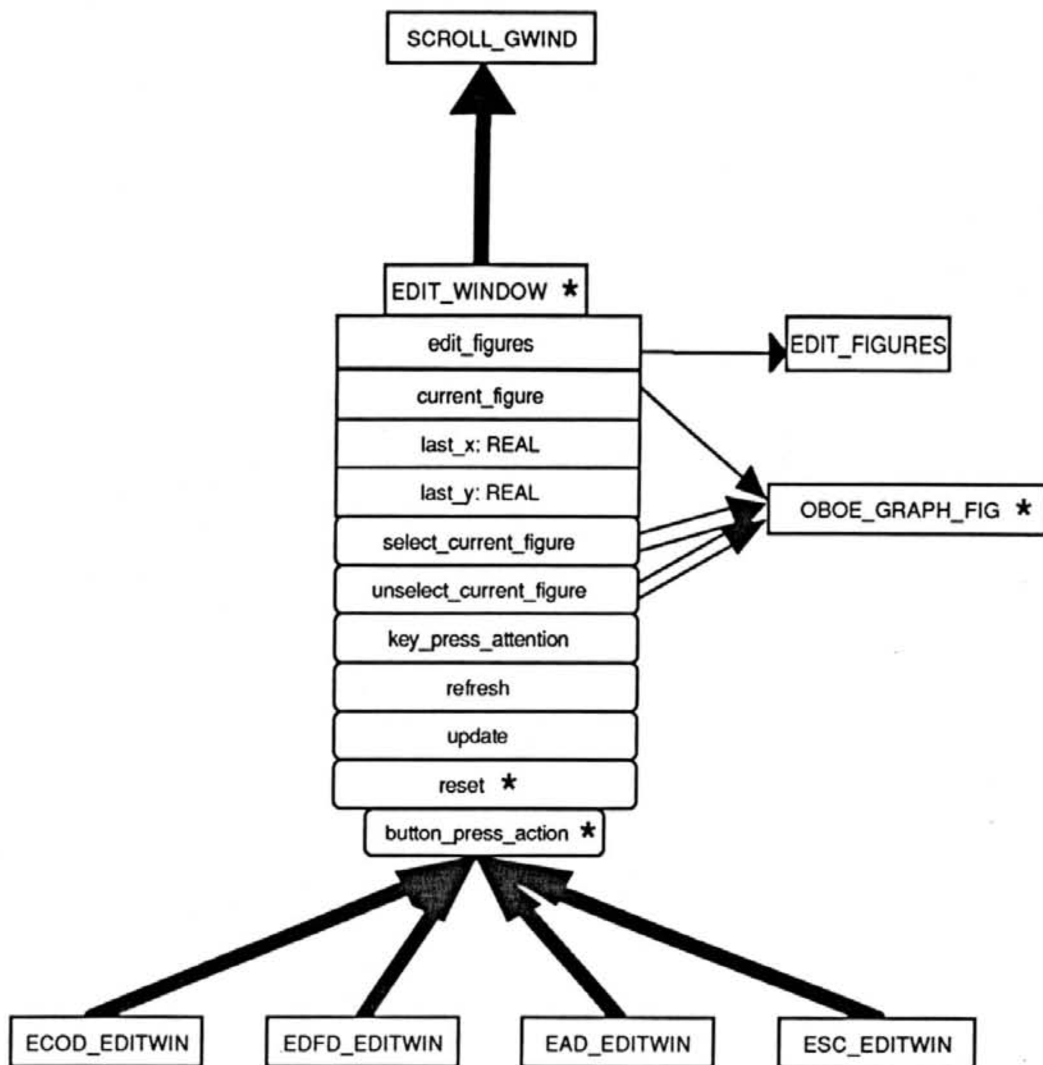


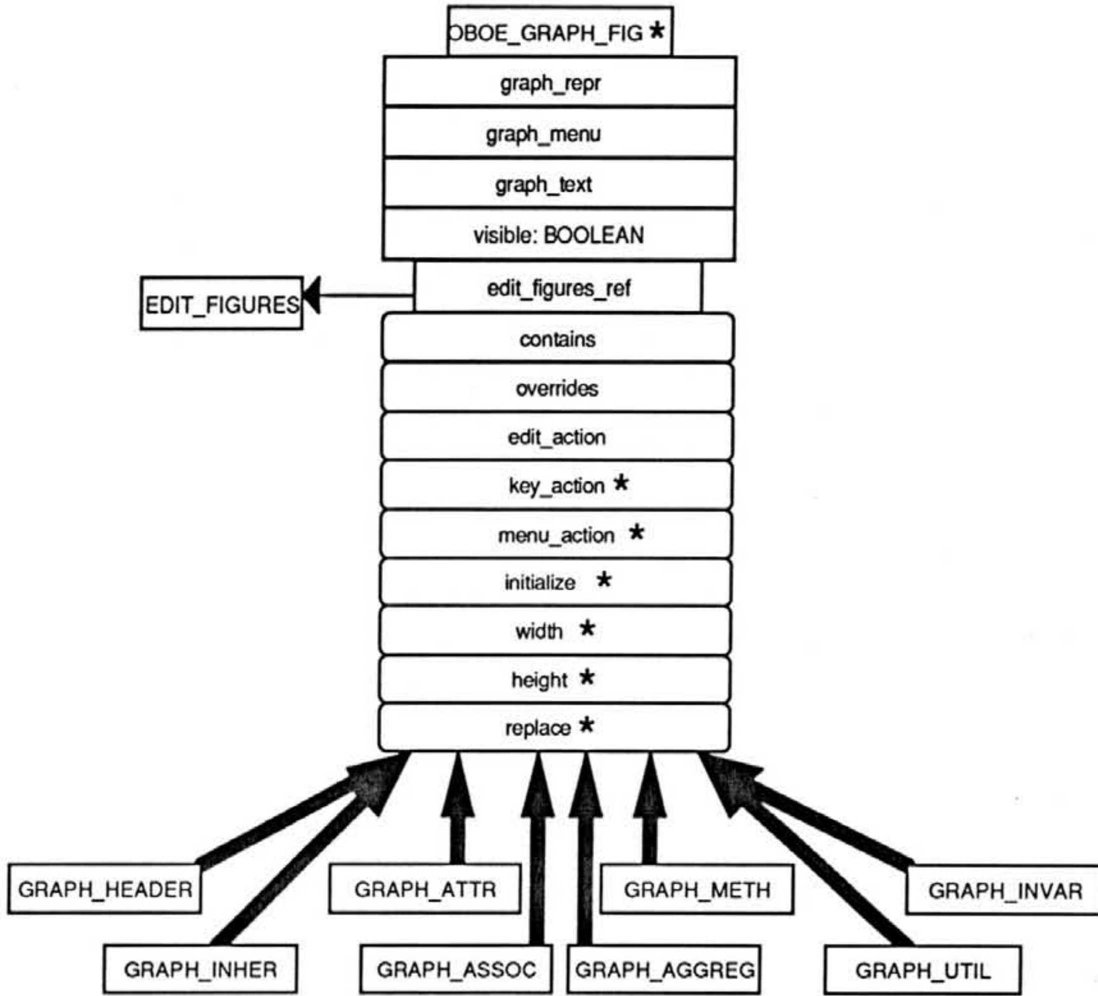



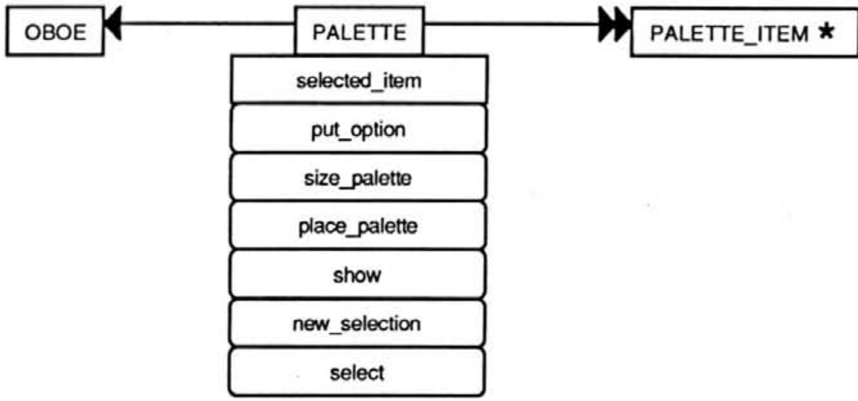

GRAPH_WINDOW
EDIT_WINDOW
SYSTEM
OBOE *
MESSAGE_LINE
PALETTE
ECOD
EDFD
ESC
EAD

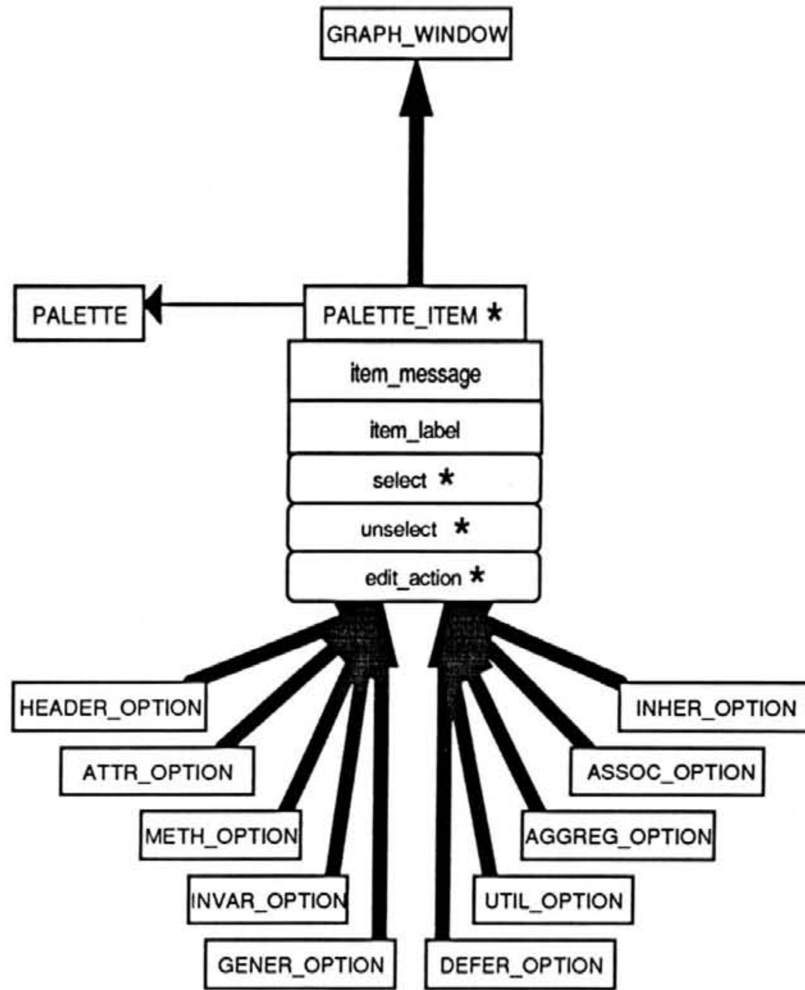

GRAPH_WINDOW
MESSAGE_LINE
message
warning_message
present
present_warning

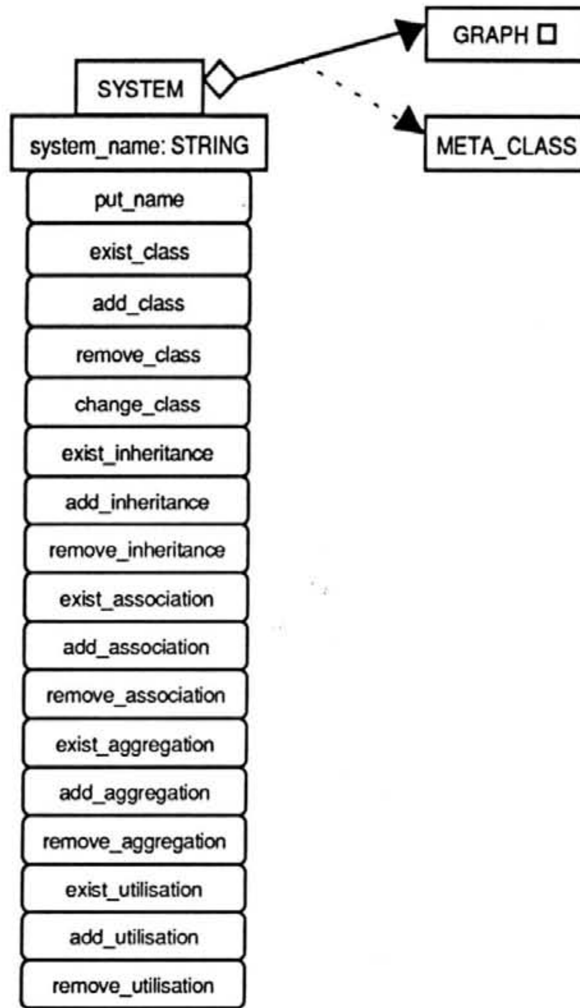

EDIT_WINDOW
EDIT_FIGURES
OBOE_GRAPH_FIG *
contains
remove_figure
add_figure
reset
update

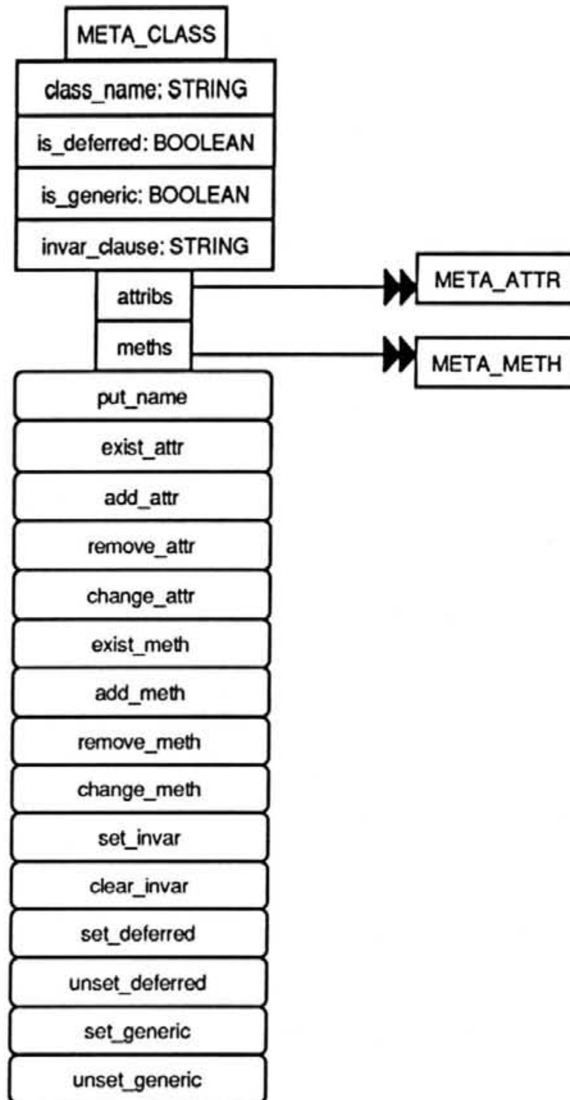

SCROLL_GWIND
EDIT_WINDOW *
edit_figures
current_figure
last_x: REAL
last_y: REAL
select_current_figure
unselect_current_figure
key_press_attention
refresh
update
reset *
button_press_action *
EDIT_FIGURES
OBOE_GRAPH_FIG *
ECOD_EDITWIN
EDFD_EDITWIN
EAD_EDITWIN
ESC_EDITWIN

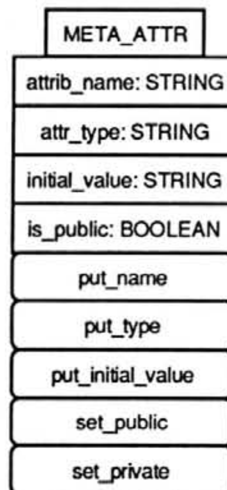

OBOE_GRAPH_FIG *
graph_repr
graph_menu
graph_text
visible: BOOLEAN
EDIT_FIGURES
edit_figures_ref
contains
overrides
edit_action
key_action *
menu_action *
initialize *
width *
height *
replace *
GRAPH_HEADER
GRAPH_ATTR
GRAPH_METH
GRAPH_INVAR
GRAPH_INHER
GRAPH_ASSOC
GRAPH_AGGREG
GRAPH_UTIL

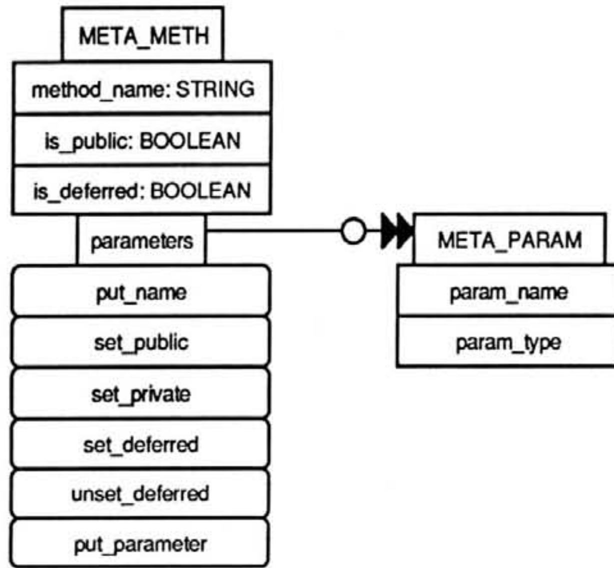

OBOE
PALETTE
PALETTE_ITEM *
selected_item
put_option
size_palette
place_palette
show
new_selection
select

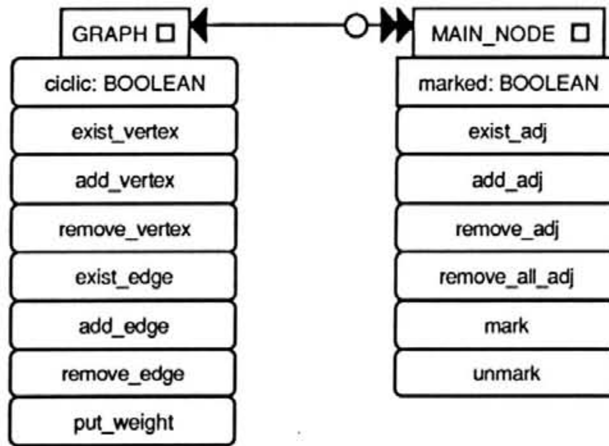

GRAPH_WINDOW
PALETTE
PALETTE_ITEM *
item_message
item_label
select *
unselect *
edit_action *
HEADER_OPTION
ATTR_OPTION
METH_OPTION
INVAR_OPTION
GENER_OPTION
INHER_OPTION
ASSOC_OPTION
AGGREG_OPTION
UTIL_OPTION
DEFER_OPTION

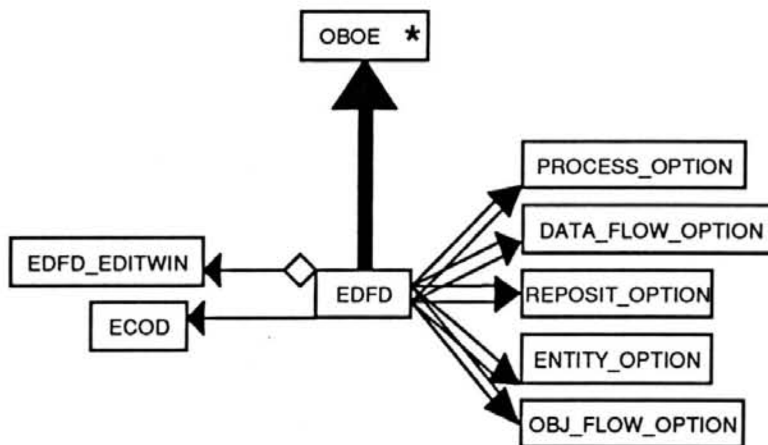

GRAPH
SYSTEM
META_CLASS
system_name: STRING
put_name
exist_class
add_class
remove_class
change_class
exist_inheritance
add_inheritance
remove_inheritance
exist_association
add_association
remove_association
exist_aggregation
add_aggregation
remove_aggregation
exist_utilisation
add_utilisation
remove_utilisation

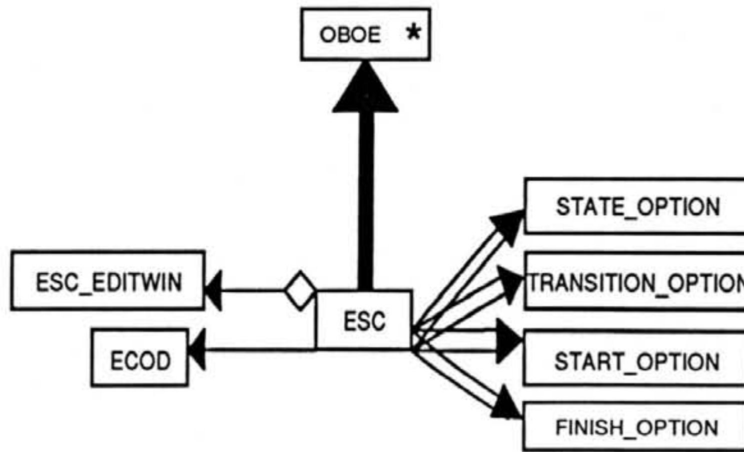

META_CLASS
class_name: STRING
is_deferred: BOOLEAN
is_generic: BOOLEAN
invar_clause: STRING
attribs
meths
META_ATTR
META_METH
put_name
exist_attr
add_attr
remove_attr
change_attr
exist_meth
add_meth
remove_meth
change_meth
set_invar
clear_invar
set_deferred
unset_deferred
set_generic
unset_generic

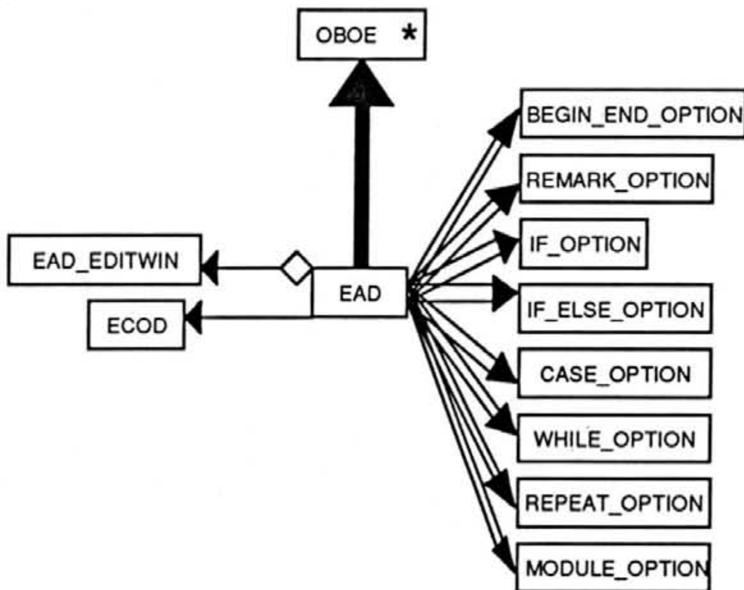

META_ATTR
attrib_name: STRING
attr_type: STRING
initial_value: STRING
is_public: BOOLEAN
put_name
put_type
put_initial_value
set_public
set_private


META_METH
method_name: STRING
is_public: BOOLEAN
is_deferred: BOOLEAN
parameters
put_name
set_public
set_private
set_deferred
unset_deferred
put_parameter
META_PARAM
param_name
param_type

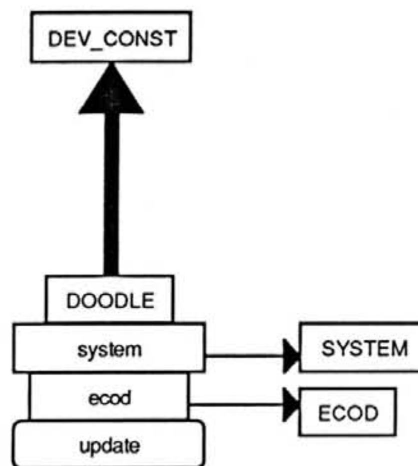

GRAPH
ciclic: BOOLEAN
exist_vertex
add_vertex
remove_vertex
exist_edge
add_edge
remove_edge
put_weight
MAIN_NODE
marked: BOOLEAN
exist_adj
add_adj
remove_adj
remove_all_adj
mark
unmark

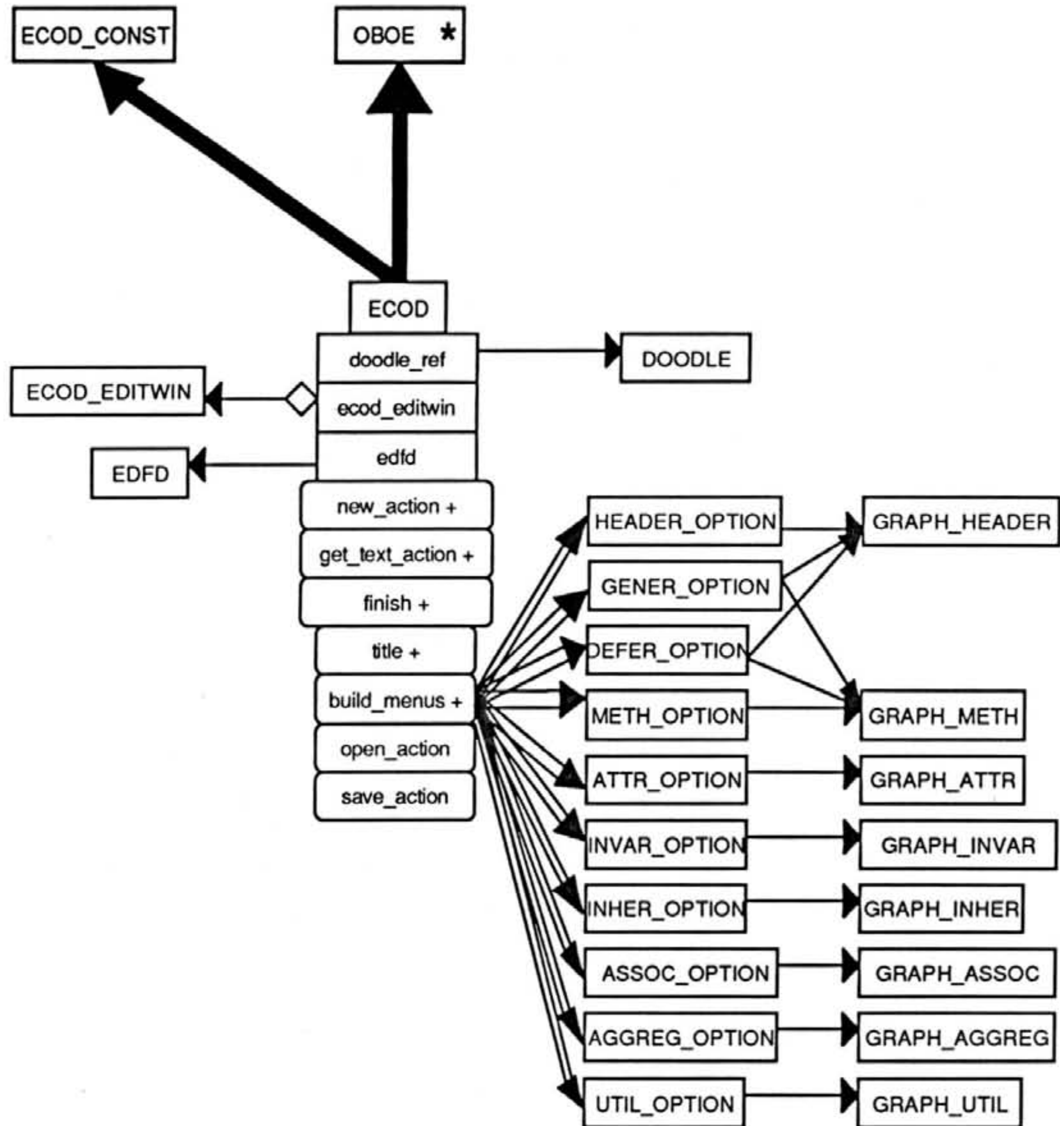

OBOE *
PROCESS_OPTION
DATA_FLOW_OPTION
EDFD_EDITWIN
EDFD
REPOSIT_OPTION
ECOD
ENTITY_OPTION
OBJ_FLOW_OPTION


OBOE *
STATE_OPTION
ESC_EDITWIN
TRANSITION_OPTION
ESC
ECOD
START_OPTION
FINISH_OPTION

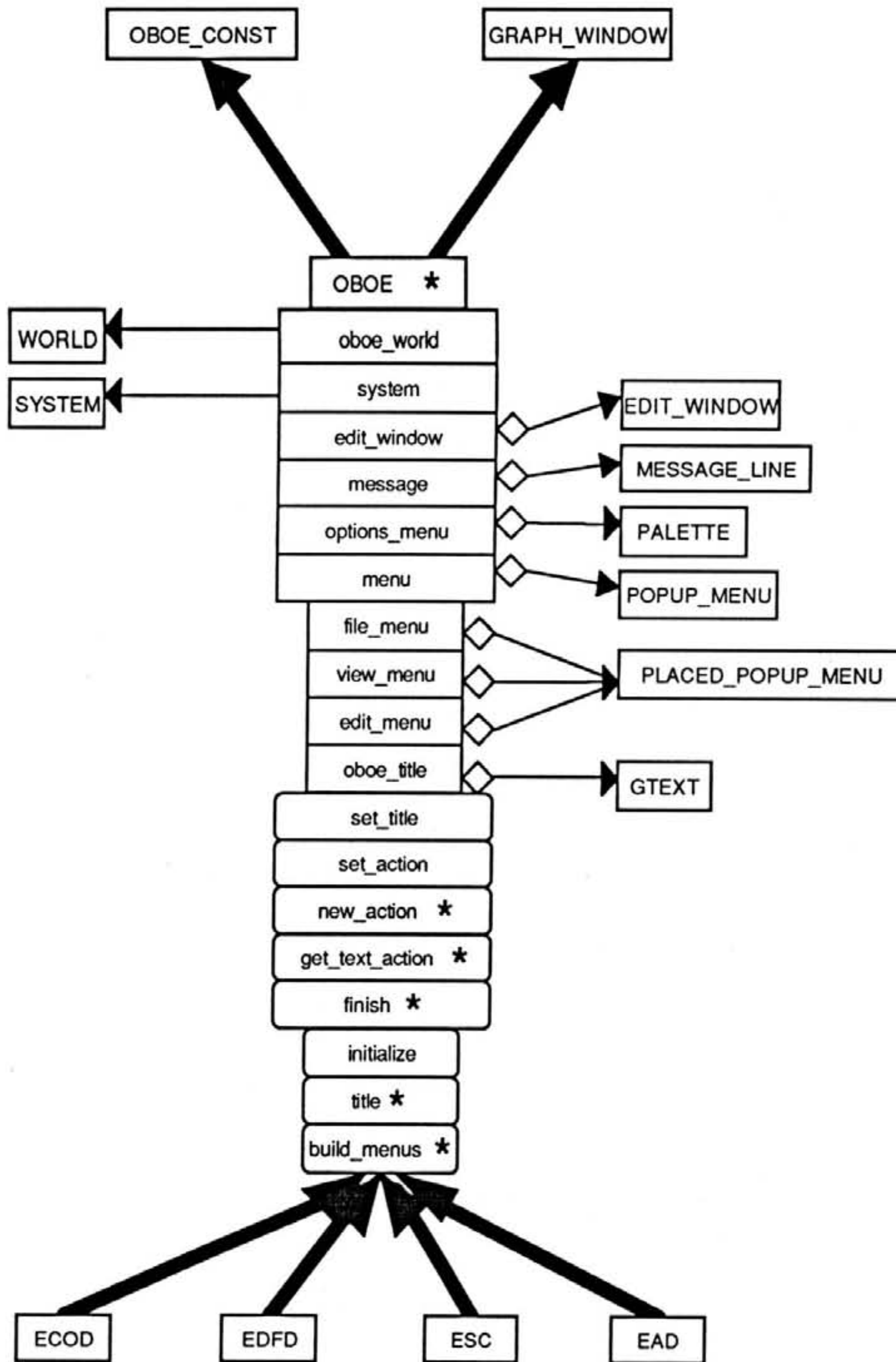

OBOE *
BEGIN_END_OPTION
REMARK_OPTION
IF_OPTION
EAD_EDITWIN
EAD
IF_ELSE_OPTION
ECOD
CASE_OPTION
WHILE_OPTION
REPEAT_OPTION
MODULE_OPTION

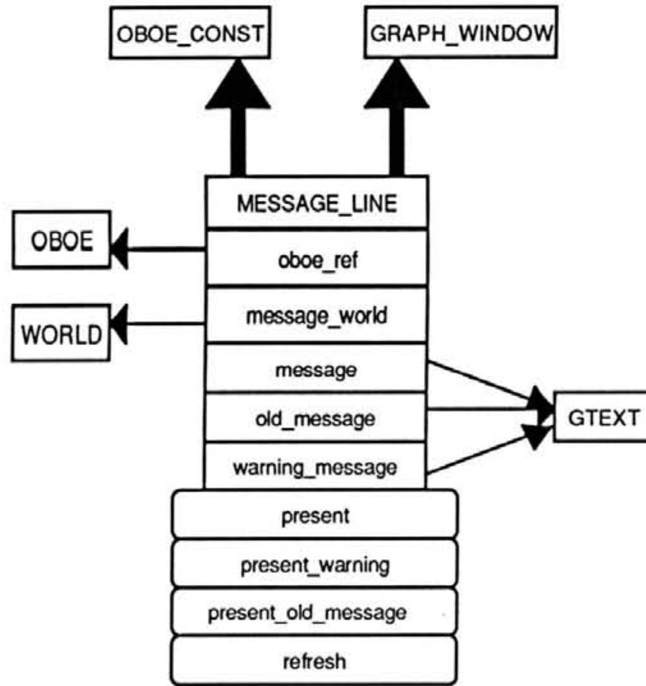
# ANEXO 3 Diagramas de classes e objetos para definição do modelo estático do sistema (Projeto).

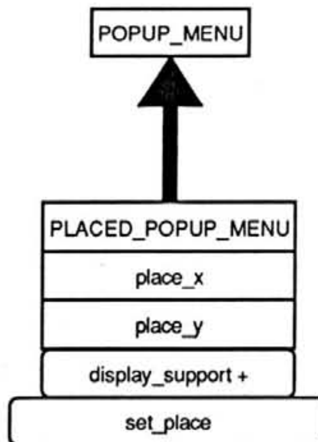


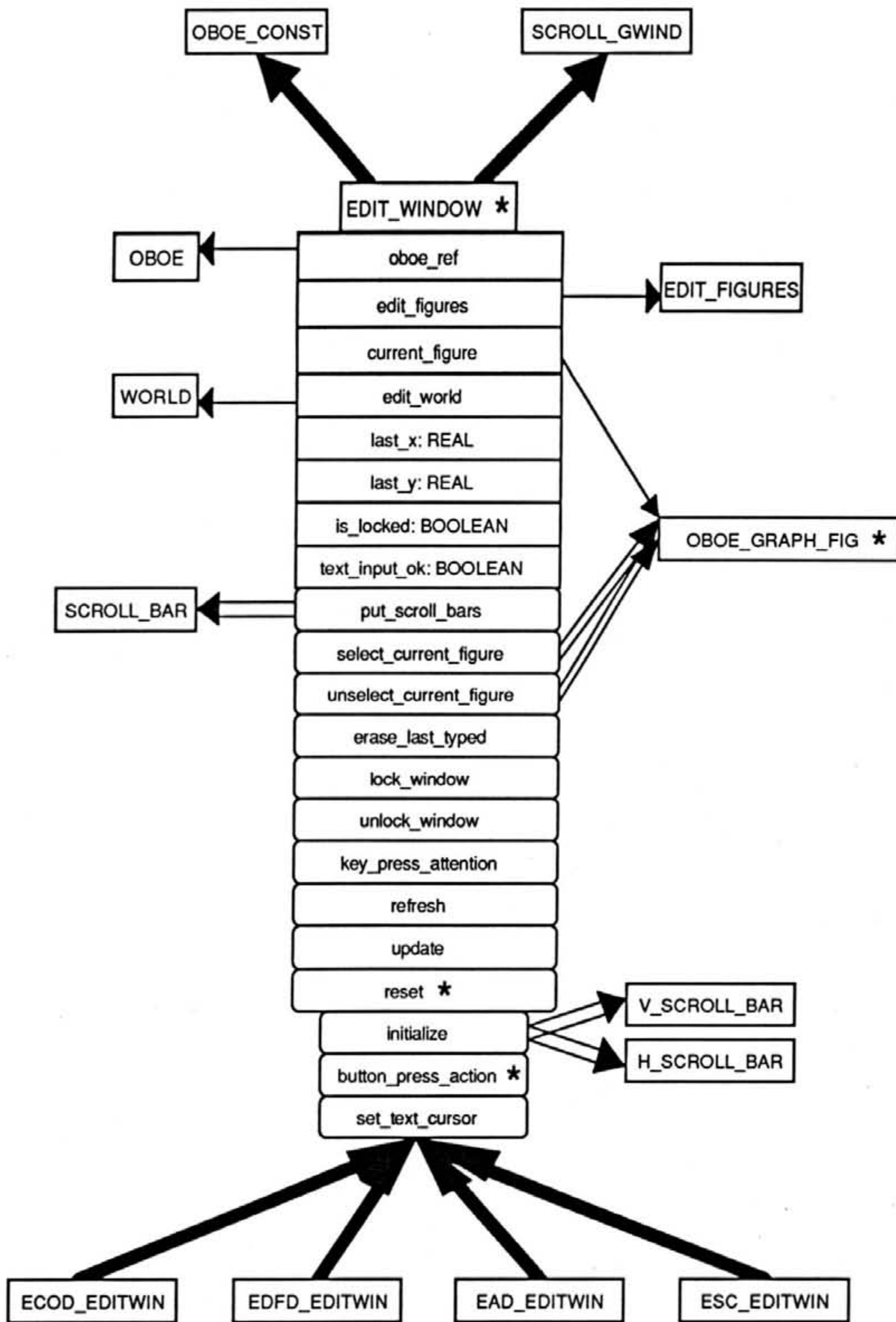


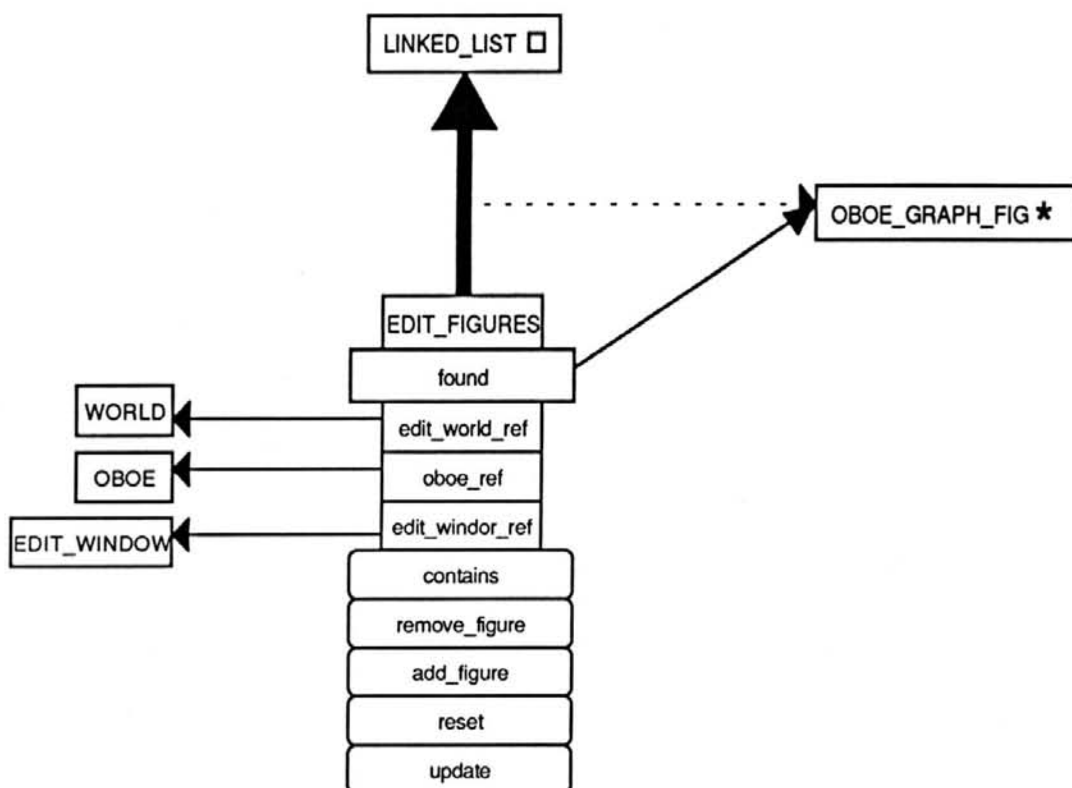
ECOD_CONST
OBOE *
ECOD
doodle_ref
ecod_editwin
edfd
new_action +
get_text_action +
finish +
title +
build_menus +
open_action
save_action
DOODLE
ECOD_EDITWIN
EDFD
HEADER_OPTION
GENER_OPTION
DEFER_OPTION
METH_OPTION
ATTR_OPTION
INVAR_OPTION
INHER_OPTION
ASSOC_OPTION
AGGREG_OPTION
UTIL_OPTION
GRAPH_HEADER
GRAPH_METH
GRAPH_ATTR
GRAPH_INVAR
GRAPH_INHER
GRAPH_ASSOC
GRAPH_AGGREG
GRAPH_UTIL

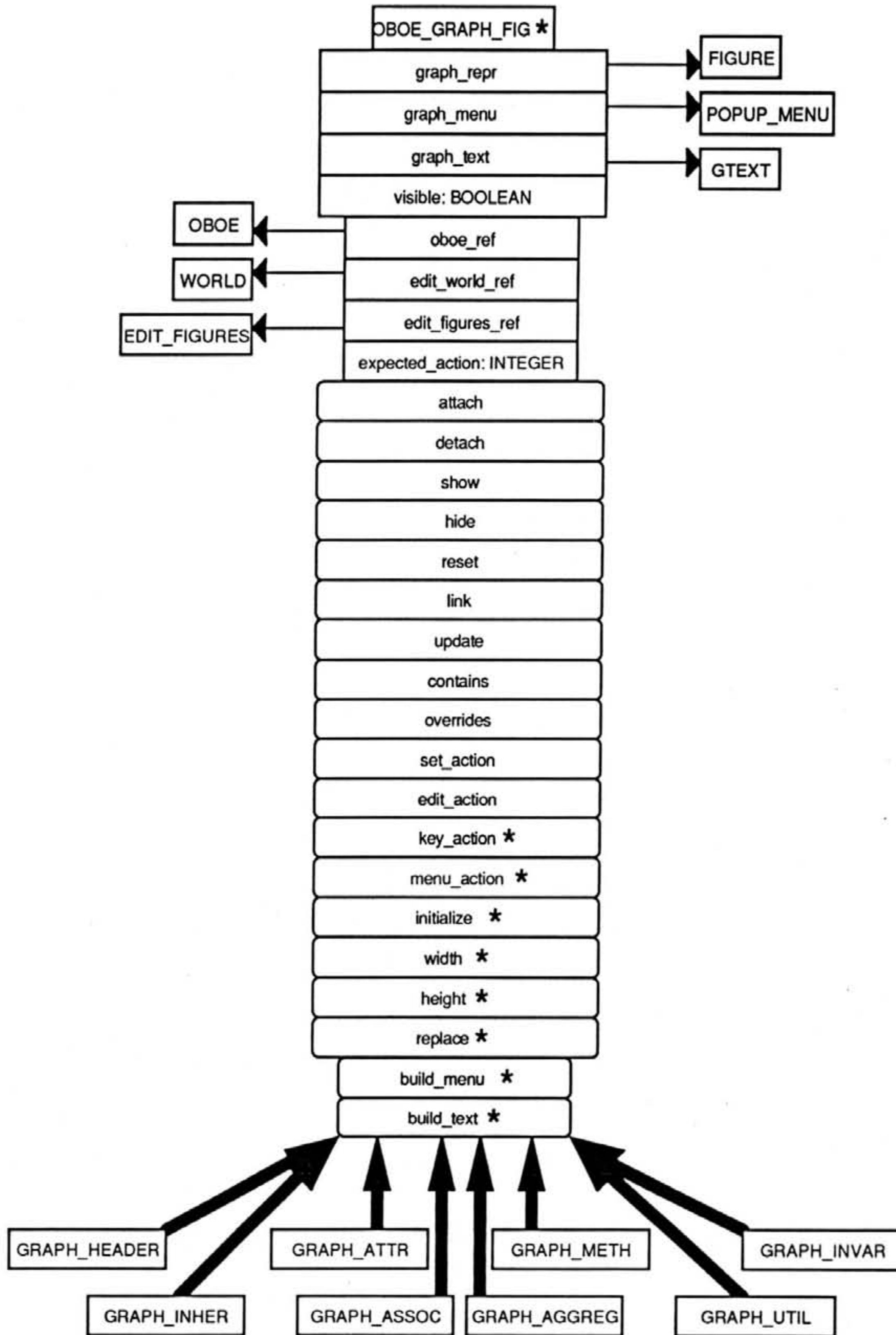

OBOE_CONST
GRAPH_WINDOW
OBOE *
WORLD
SYSTEM
oboe_world
system
edit_window
message
options_menu
menu
file_menu
view_menu
edit_menu
oboe_title
set_title
set_action
new_action *
get_text_action *
finish *
initialize
title *
build_menus *
EDIT_WINDOW
MESSAGE_LINE
PALETTE
POPUP_MENU
PLACED_POPUP_MENU
GTEXT
ECOD
EDFD
ESC
EAD

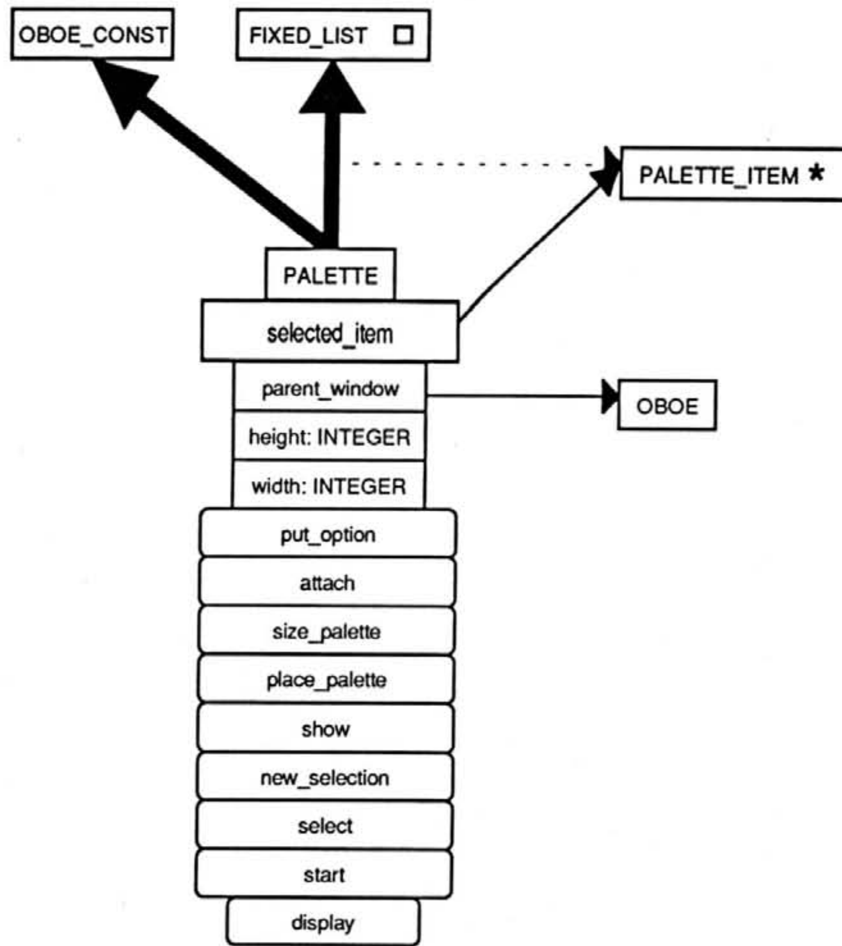

OBOE_CONST
GRAPH_WINDOW
MESSAGE_LINE
OBOE
oboe_ref
WORLD
message_world
message
old_message
GTEXT
warning_message
present
present_warning
present_old_message
refresh

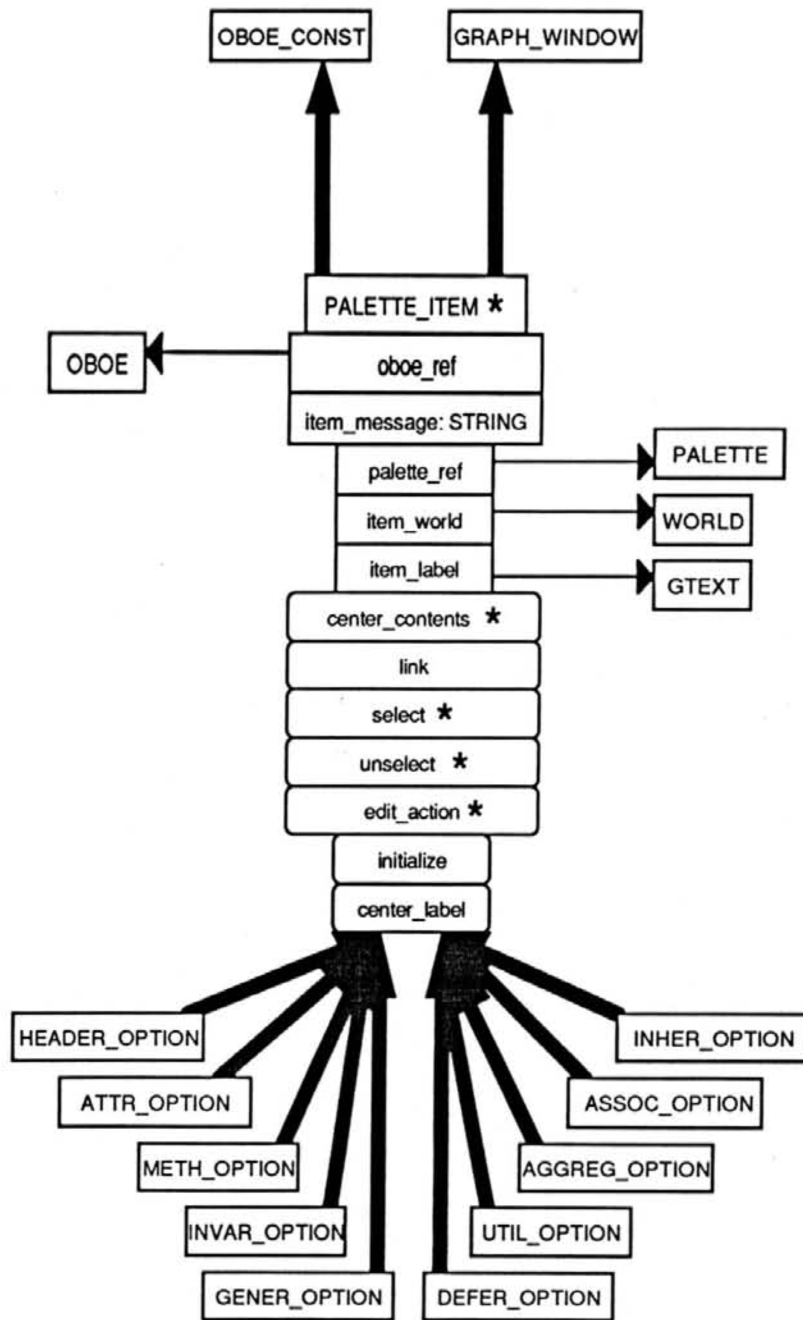

POPUP_MENU
PLACED_POPUP_MENU
place_x
place_y
display_support +
set_place

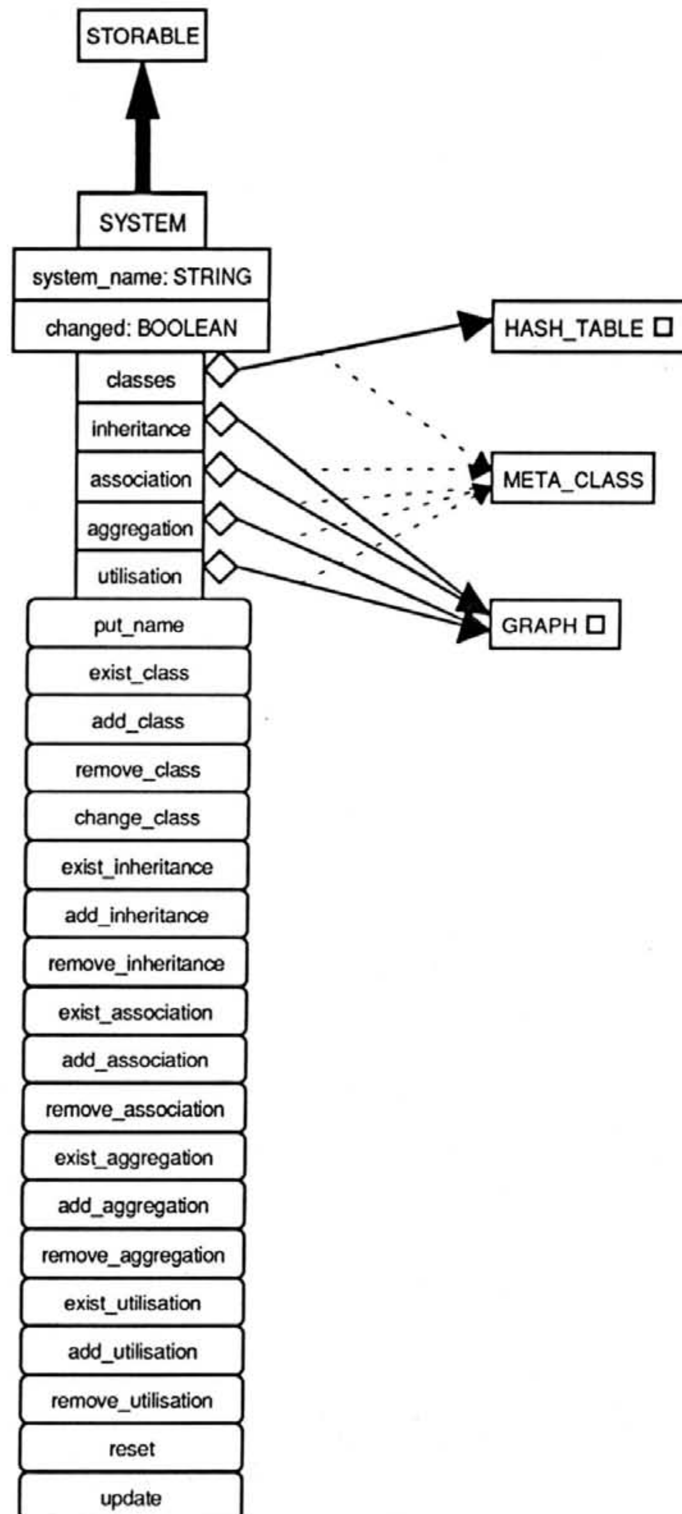

OBOE_CONST
SCROLL_GWIND
EDIT_WINDOW *
OBOE
oboe_ref
edit_figures
EDIT_FIGURES
current_figure
WORLD
edit_world
last_x: REAL
last_y: REAL
is_locked: BOOLEAN
OBOE_GRAPH_FIG *
text_input_ok: BOOLEAN
SCROLL_BAR
put_scroll_bars
select_current_figure
unselect_current_figure
erase_last_typed
lock_window
unlock_window
key_press_attention
refresh
update
reset *
V_SCROLL_BAR
initialize
H_SCROLL_BAR
button_press_action *
set_text_cursor
ECOD_EDITWIN
EDFD_EDITWIN
EAD_EDITWIN
ESC_EDITWIN


LINKED_LIST
OBOE_GRAPH_FIG *
EDIT_FIGURES
found
WORLD
edit_world_ref
OBOE
oboe_ref
EDIT_WINDOW
edit_windor_ref
contains
remove_figure
add_figure
reset
update

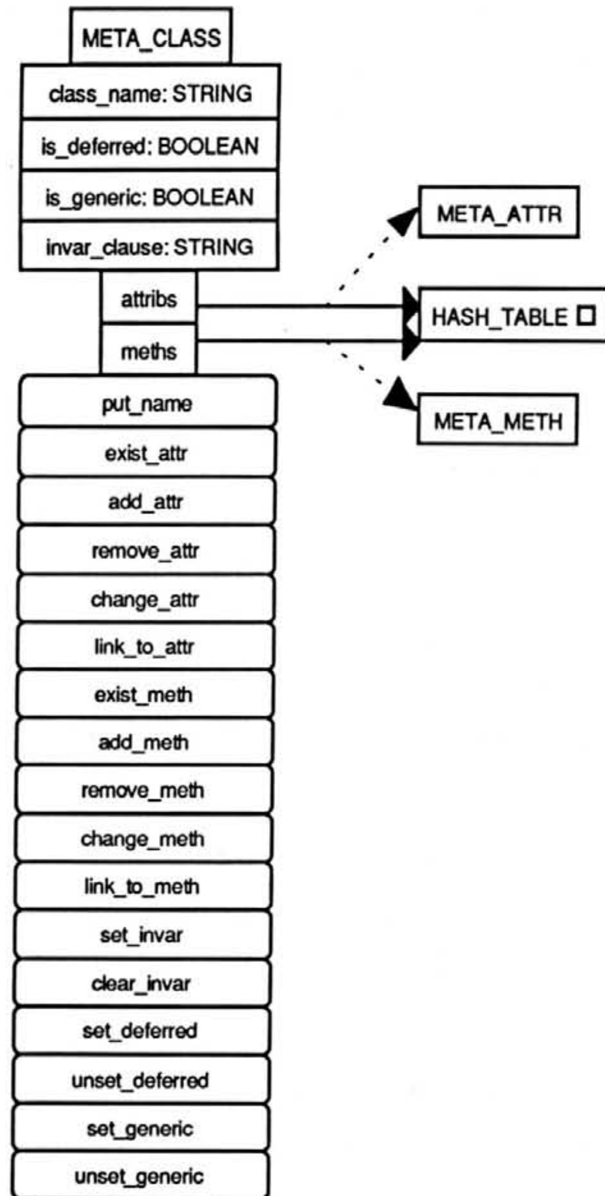

OBOE_GRAPH_FIG *
graph_repr
graph_menu
graph_text
visible: BOOLEAN
oboe_ref
edit_world_ref
edit_figures_ref
expected_action: INTEGER
attach
detach
show
hide
reset
link
update
contains
overrides
set_action
edit_action
key_action *
menu_action *
initialize *
width *
height *
replace *
build_menu *
build_text *
FIGURE
POPUP_MENU
GTEXT
OBOE
WORLD
EDIT_FIGURES
GRAPH_HEADER
GRAPH_ATTR
GRAPH_METH
GRAPH_INVAR
GRAPH_INHER
GRAPH_ASSOC
GRAPH_AGGREG
GRAPH_UTIL

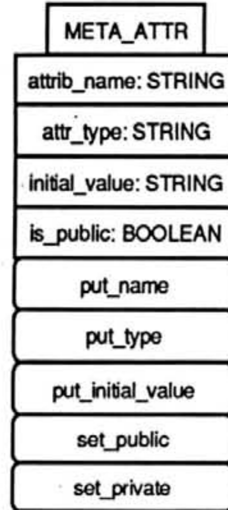

OBOE_CONST
FIXED_LIST
PALETTE_ITEM *
PALETTE
selected_item
parent_window
OBOE
height: INTEGER
width: INTEGER
put_option
attach
size_palette
place_palette
show
new_selection
select
start
display

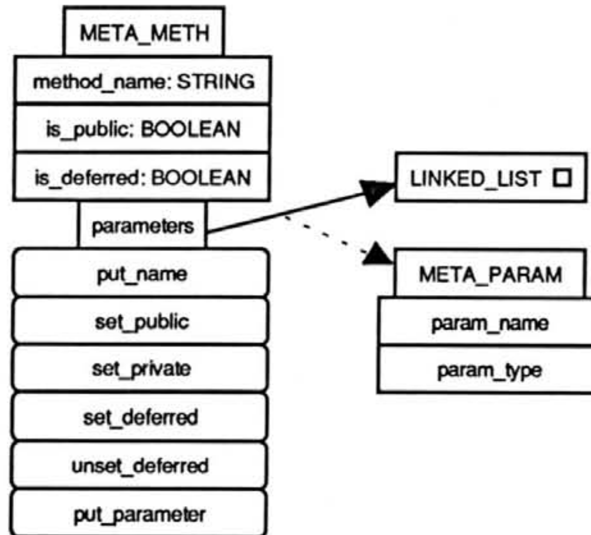

OBOE_CONST
GRAPH_WINDOW
PALETTE_ITEM *
OBOE
oboe_ref
item_message: STRING
palette_ref
PALETTE
item_world
WORLD
item_label
GTEXT
center_contents *
link
select *
unselect *
edit_action *
initialize
center_label
HEADER_OPTION
ATTR_OPTION
METH_OPTION
INVAR_OPTION
GENER_OPTION
INHER_OPTION
ASSOC_OPTION
AGGREG_OPTION
UTIL_OPTION
DEFER_OPTION

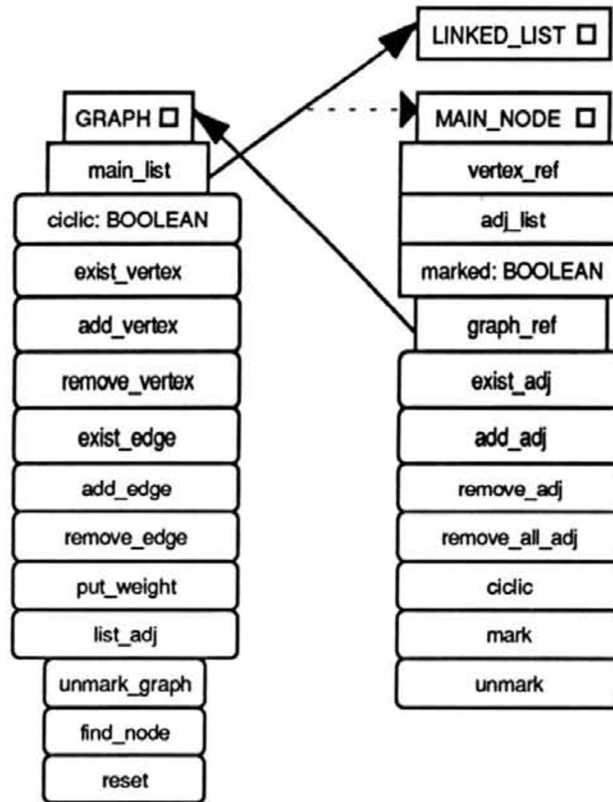


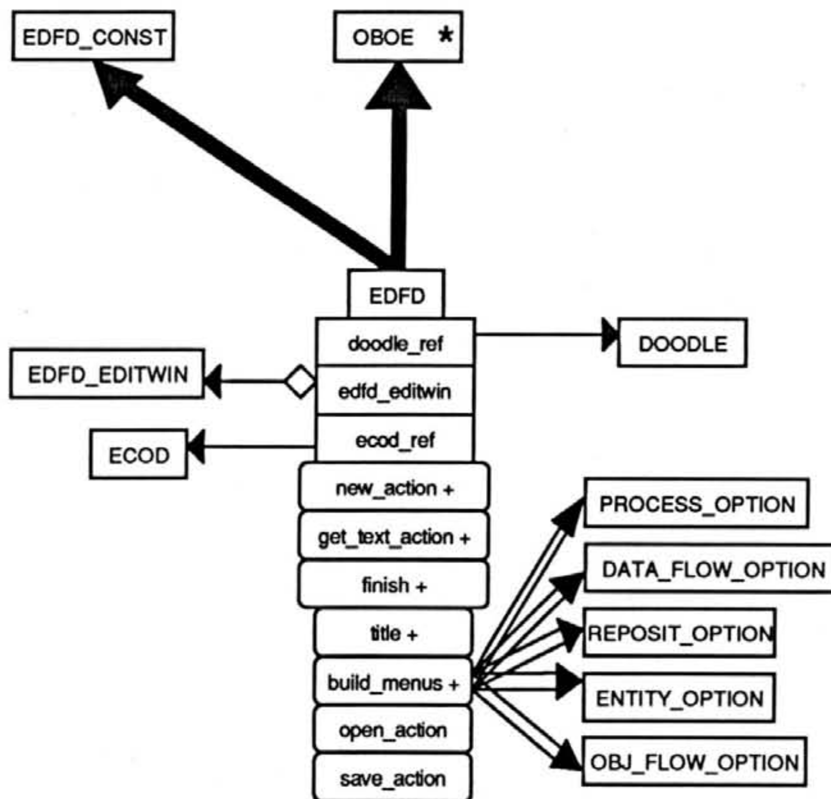
STORABLE
SYSTEM
system_name: STRING
changed: BOOLEAN
classes
inheritance
association
aggregation
utilisation
put_name
exist_class
add_class
remove_class
change_class
exist_inheritance
add_inheritance
remove_inheritance
exist_association
add_association
remove_association
exist_aggregation
add_aggregation
remove_aggregation
exist_utilisation
add_utilisation
remove_utilisation
reset
update
HASH_TABLE
META_CLASS
GRAPH

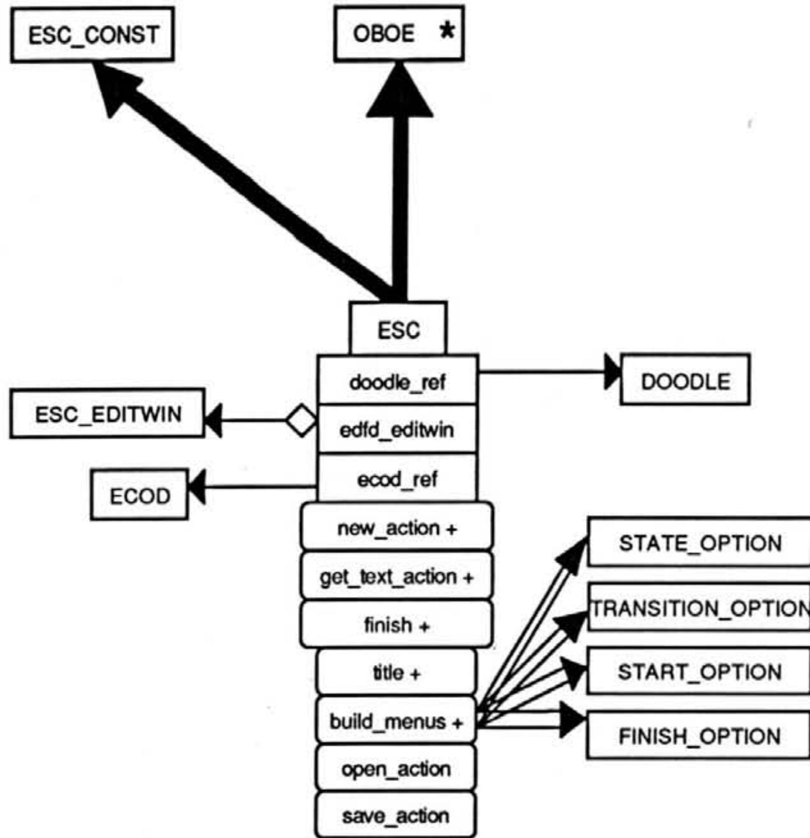

META_CLASS
class_name: STRING
is_deferred: BOOLEAN
is_generic: BOOLEAN
invar_clause: STRING
attribs
meths
META_ATTR
HASH_TABLE
META_METH
put_name
exist_attr
add_attr
remove_attr
change_attr
link_to_attr
exist_meth
add_meth
remove_meth
change_meth
link_to_meth
set_invar
clear_invar
set_deferred
unset_deferred
set_generic
unset_generic

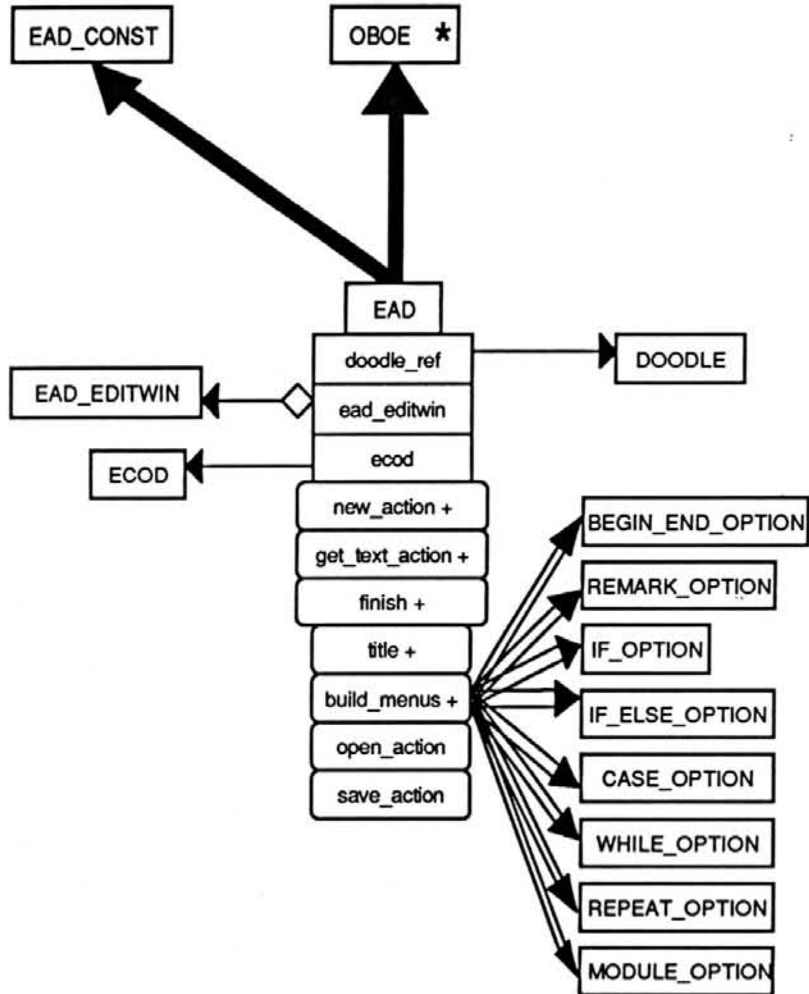

META_ATTR
attrib_name: STRING
attr_type: STRING
initial_value: STRING
is_public: BOOLEAN
put_name
put_type
put_initial_value
set_public
set_private


META_METH
method_name: STRING
is_public: BOOLEAN
is_deferred: BOOLEAN
parameters
put_name
set_public
set_private
set_deferred
unset_deferred
put_parameter
LINKED_LIST
META_PARAM
param_name
param_type

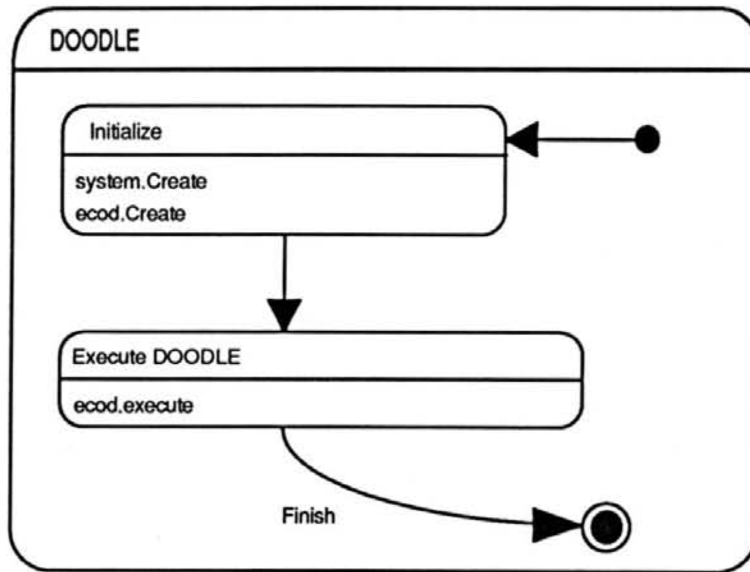

LINKED_LIST
GRAPH
MAIN_NODE
main_list
ciclic: BOOLEAN
exist_vertex
add_vertex
remove_vertex
exist_edge
add_edge
remove_edge
put_weight
list_adj
unmark_graph
find_node
reset
vertex_ref
adj_list
marked: BOOLEAN
graph_ref
exist_adj
add_adj
remove_adj
remove_all_adj
ciclic
mark
unmark

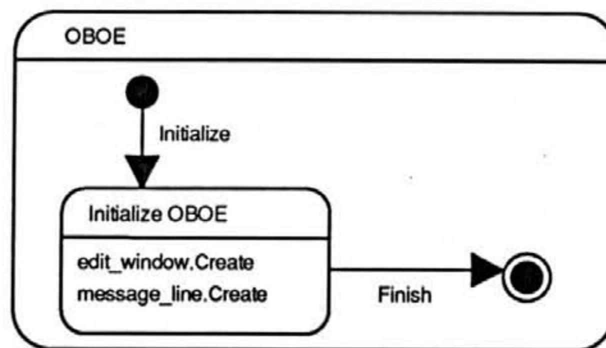

EDFD_CONST
OBOE *
EDFD
doodle_ref
edfd_editwin
ecod_ref
new_action +
get_text_action +
finish +
title +
build_menus +
open_action
save_action
DOODLE
EDFD_EDITWIN
ECOD
PROCESS_OPTION
DATA_FLOW_OPTION
REPOSIT_OPTION
ENTITY_OPTION
OBJ_FLOW_OPTION

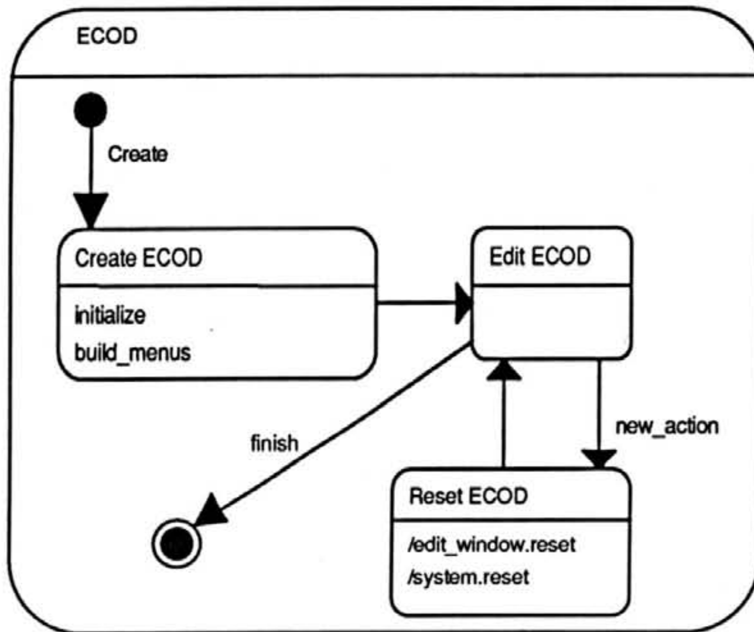

ESC_CONST
OBOE *
ESC
doodle_ref
DOODLE
ESC_EDITWIN
edfd_editwin
ECOD
ecod_ref
new_action +
get_text_action +
finish +
title +
build_menus +
open_action
save_action
STATE_OPTION
TRANSITION_OPTION
START_OPTION
FINISH_OPTION

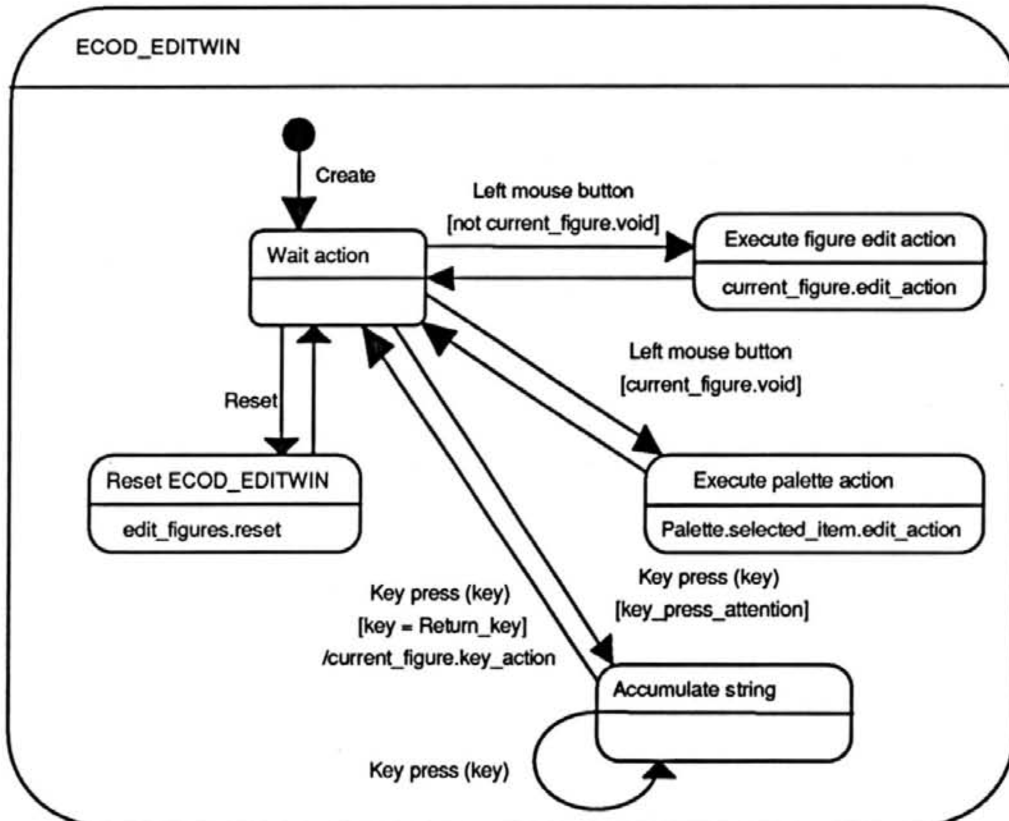

EAD_CONST
OBOE *
EAD
doodle_ref
DOODLE
EAD_EDITWIN
ead_editwin
ECOD
ecod
new_action +
get_text_action +
finish +
title +
build_menus +
open_action
save_action
BEGIN_END_OPTION
REMARK_OPTION
IF_OPTION
IF_ELSE_OPTION
CASE_OPTION
WHILE_OPTION
REPEAT_OPTION
MODULE_OPTION

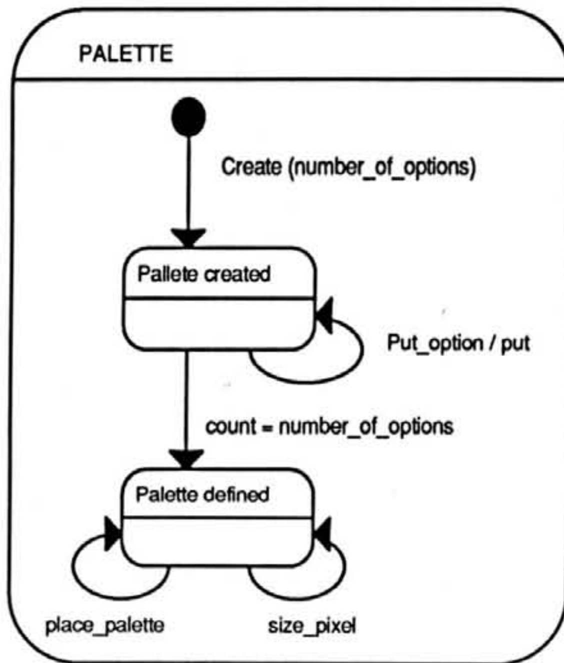
# ANEXO 4 Diagramas de Estado para a definição do modelo dinâmico do sistema.

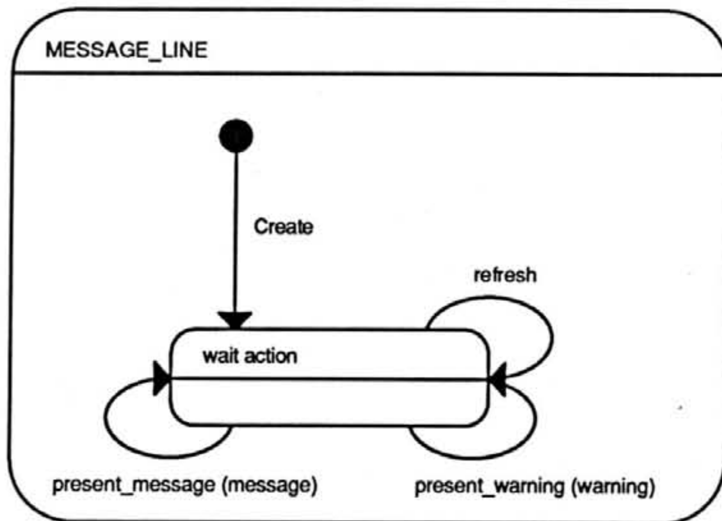


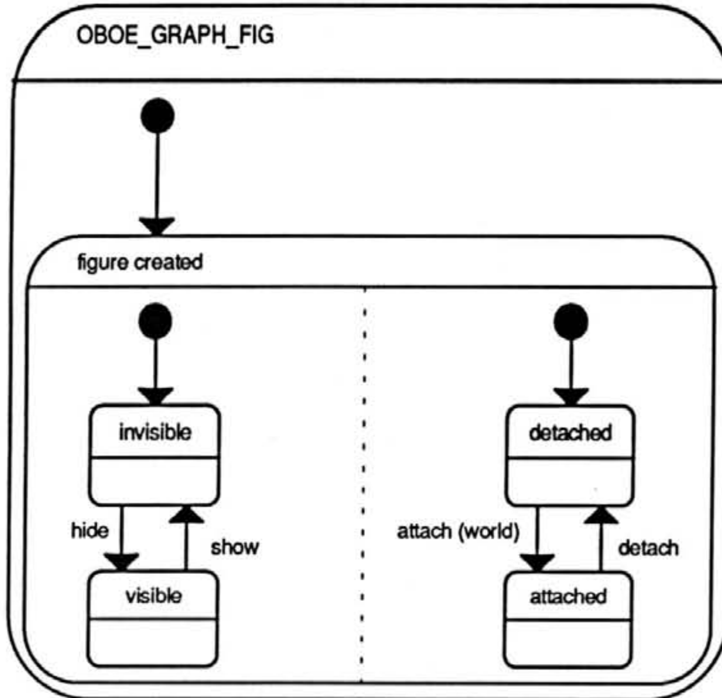



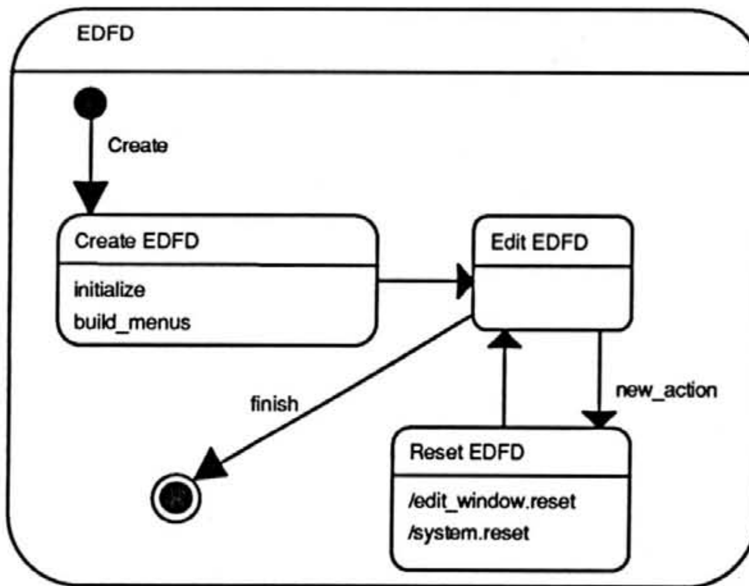

ECOD
Create
Create ECOD
initialize
build_menus
Edit ECOD
finish
new_action
Reset ECOD
/edit_window.reset
/system.reset

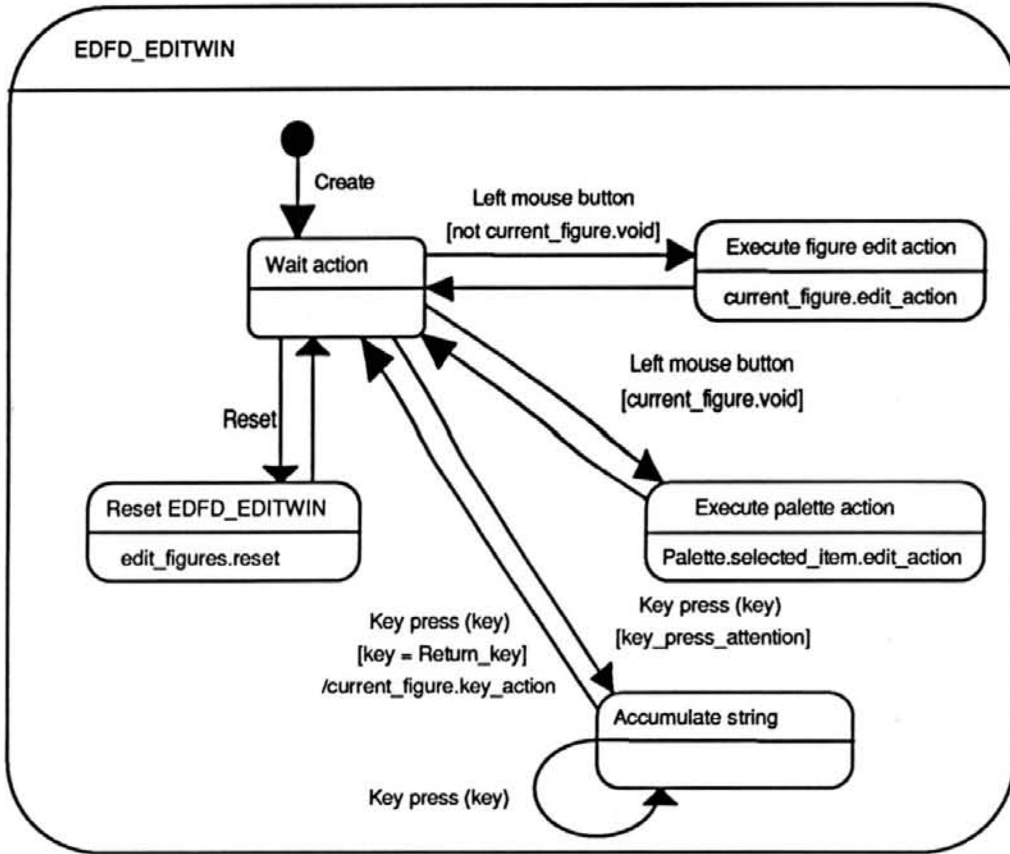

ECOD_EDITWIN
Create
Left mouse button
[not current_figure.void]
Wait action
Execute figure edit action
current_figure.edit_action
Left mouse button
[current_figure.void]
Reset
Reset ECOD_EDITWIN
edit_figures.reset
Execute palette action
Palette.selected_item.edit_action
Key press (key)
[key = Return_key]
/current_figure.key_action
Key press (key)
[key_press_attention]
Accumulate string
Key press (key)

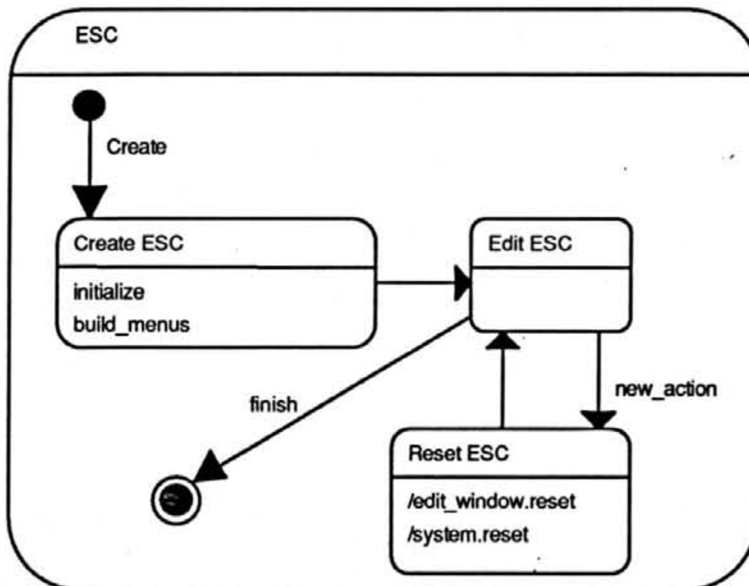

PALETTE
Create (number_of_options)
Pallete created
Put_option / put
count = number_of_options
Palette defined
place_palette
size_pixel

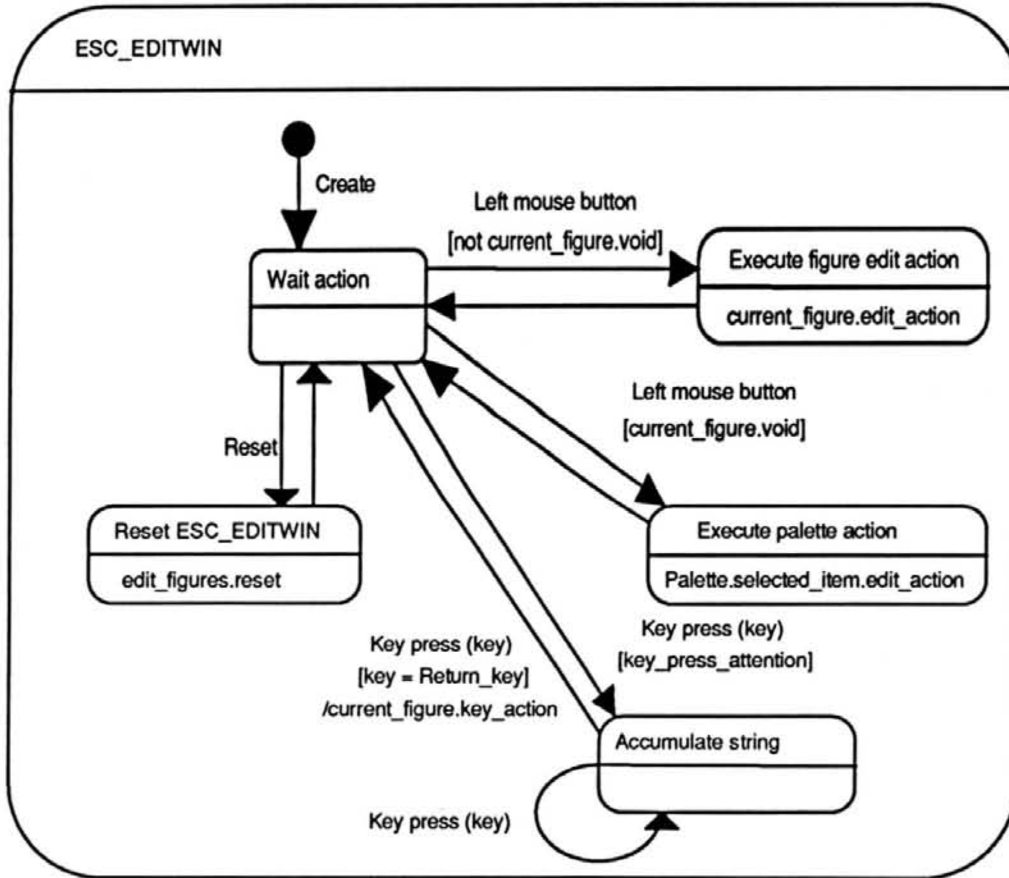

MESSAGE_LINE
Create
refresh
wait action
present_message (message)
present_warning (warning)

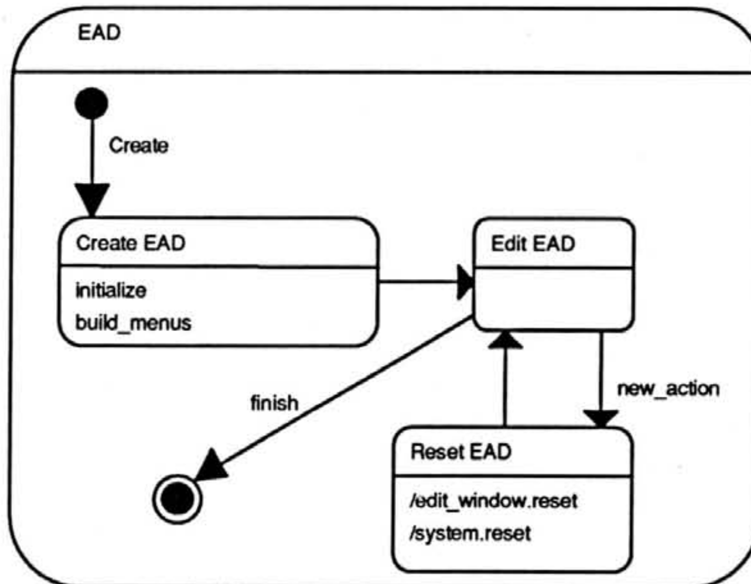

OBOE_GRAPH_FIG
figure created
invisible
hide
show
visible
detached
attach (world)
detach
attached

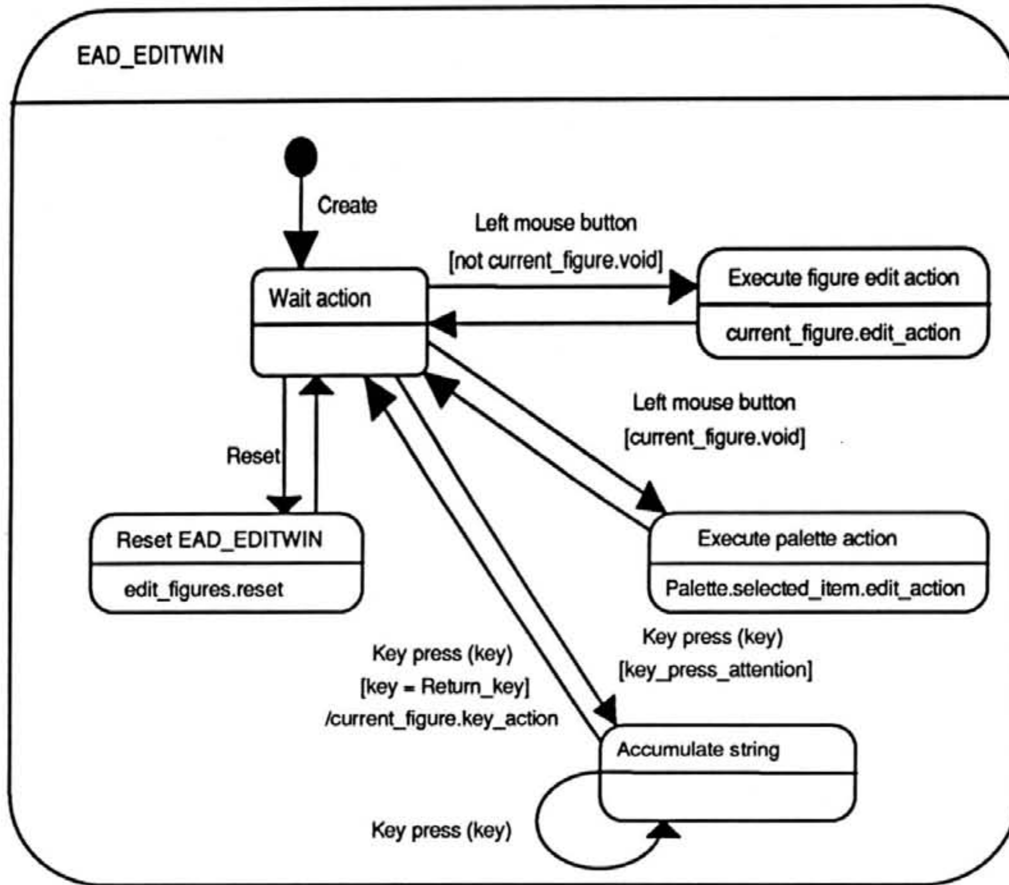

EDFD
Create
Create EDFD
initialize
build_menus
Edit EDFD
finish
new_action
Reset EDFD
/edit_window.reset
/system.reset


EDFD_EDITWIN
Create
Wait action
Left mouse button
[not current_figure.void]
Execute figure edit action
current_figure.edit_action
Left mouse button
[current_figure.void]
Reset
Reset EDFD_EDITWIN
edit_figures.reset
Execute palette action
Palette.selected_item.edit_action
Key press (key)
[key = Return_key]
/current_figure.key_action
Key press (key)
[key_press_attention]
Accumulate string
Key press (key)

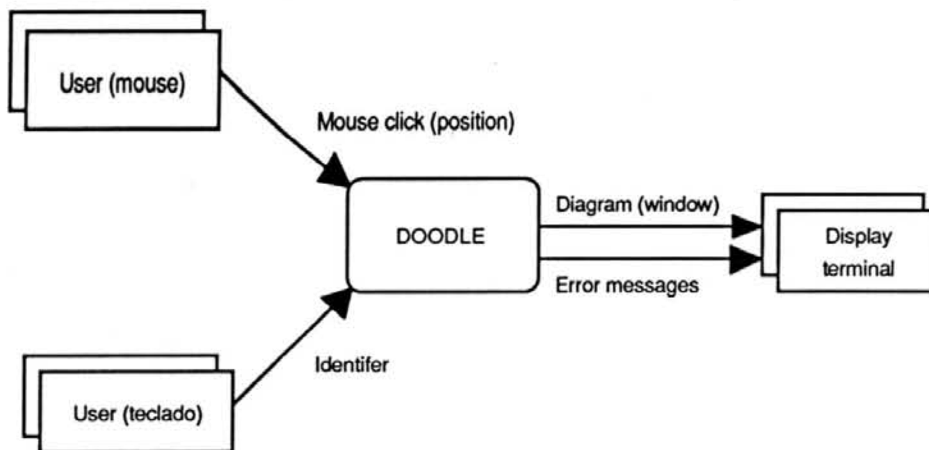

ESC
Create
Create ESC
initialize
build_menus
Edit ESC
finish
new_action
Reset ESC
/edit_window.reset
/system.reset


ESC_EDITWIN
Create
Left mouse button
[not current_figure.void]
Wait action
Execute figure edit action
current_figure.edit_action
Left mouse button
[current_figure.void]
Reset
Reset ESC_EDITWIN
edit_figures.reset
Execute palette action
Palette.selected_item.edit_action
Key press (key)
[key = Return_key]
/current_figure.key_action
Key press (key)
[key_press_attention]
Accumulate string
Key press (key)

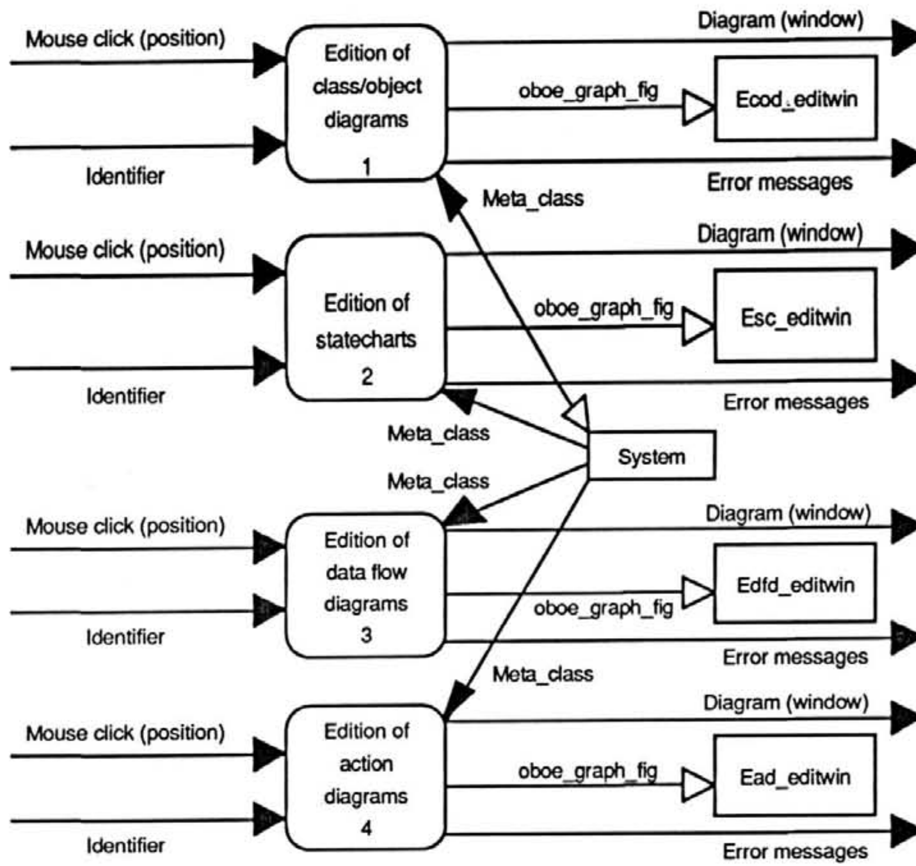

EAD
Create
Create EAD
initialize
build_menus
Edit EAD
finish
new_action
Reset EAD
/edit_window.reset
/system.reset


EAD_EDITWIN
Create
Left mouse button
[not current_figure.void]
Wait action
Execute figure edit action
current_figure.edit_action
Left mouse button
[current_figure.void]
Reset
Reset EAD_EDITWIN
edit_figures.reset
Execute palette action
Palette.selected_item.edit_action
Key press (key)
[key_press_attention]
Key press (key)
[key = Return_key]
/current_figure.key_action
Accumulate string
Key press (key)

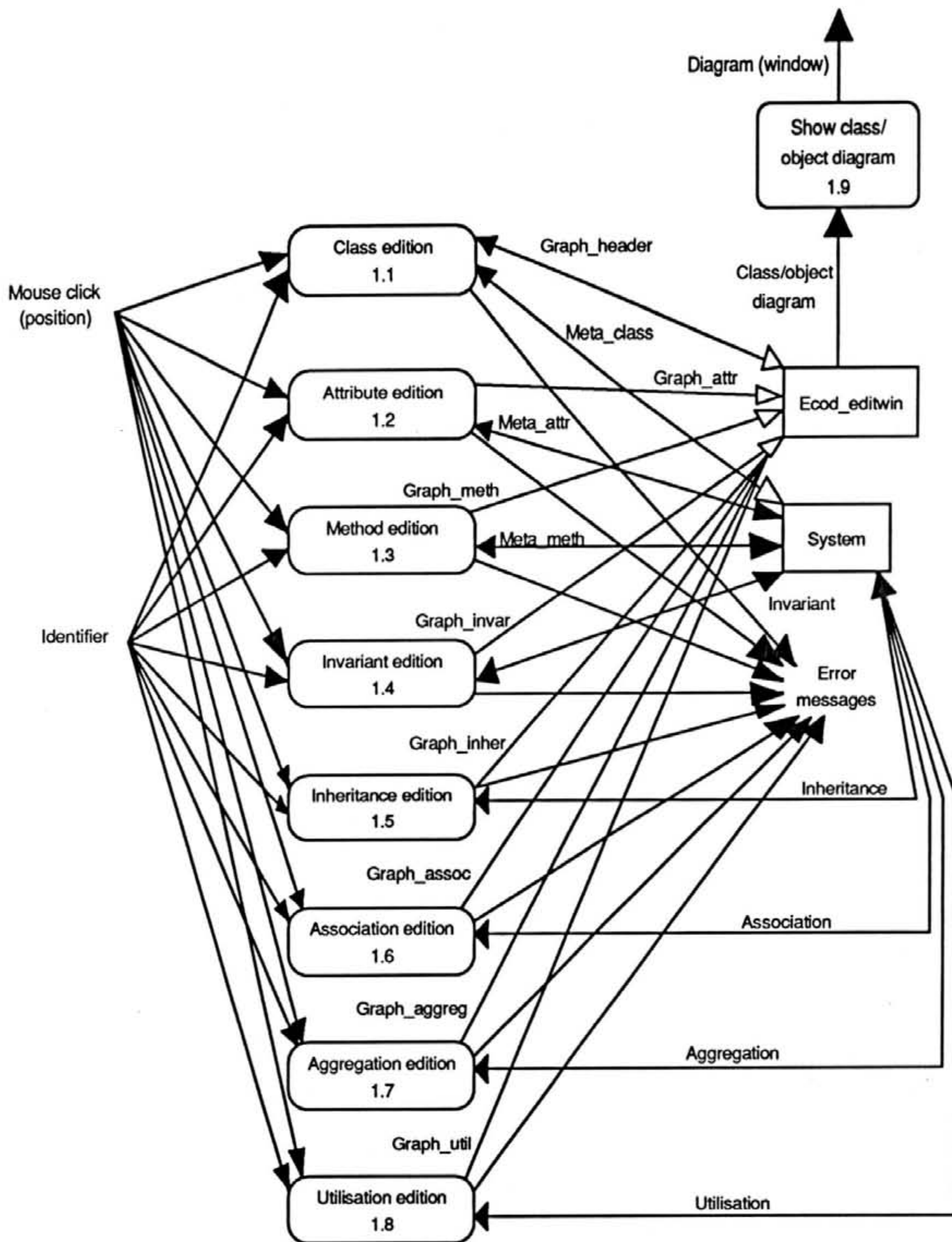
# ANEXO 5 Diagramas de Fluxo de Dados e Diagramas de Ação para a definição do modelo funcional do sistema.



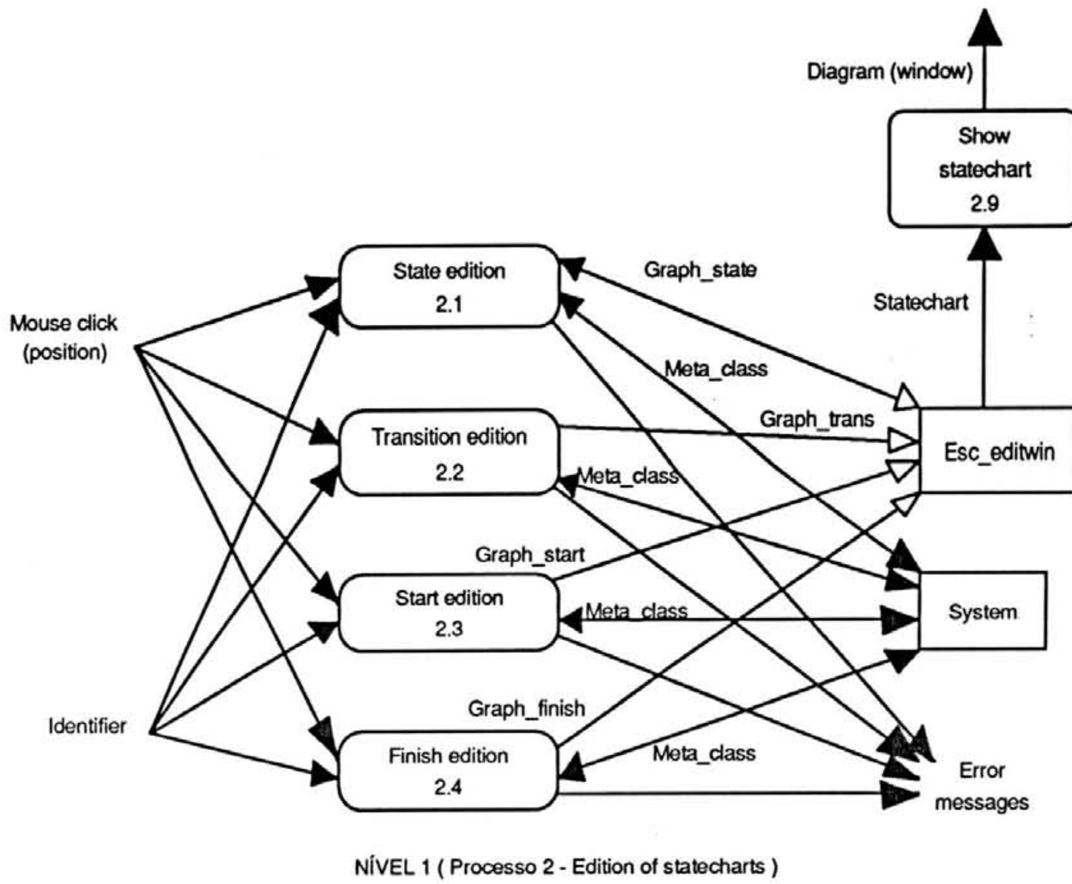

NÍVEL CONTEXTUAL



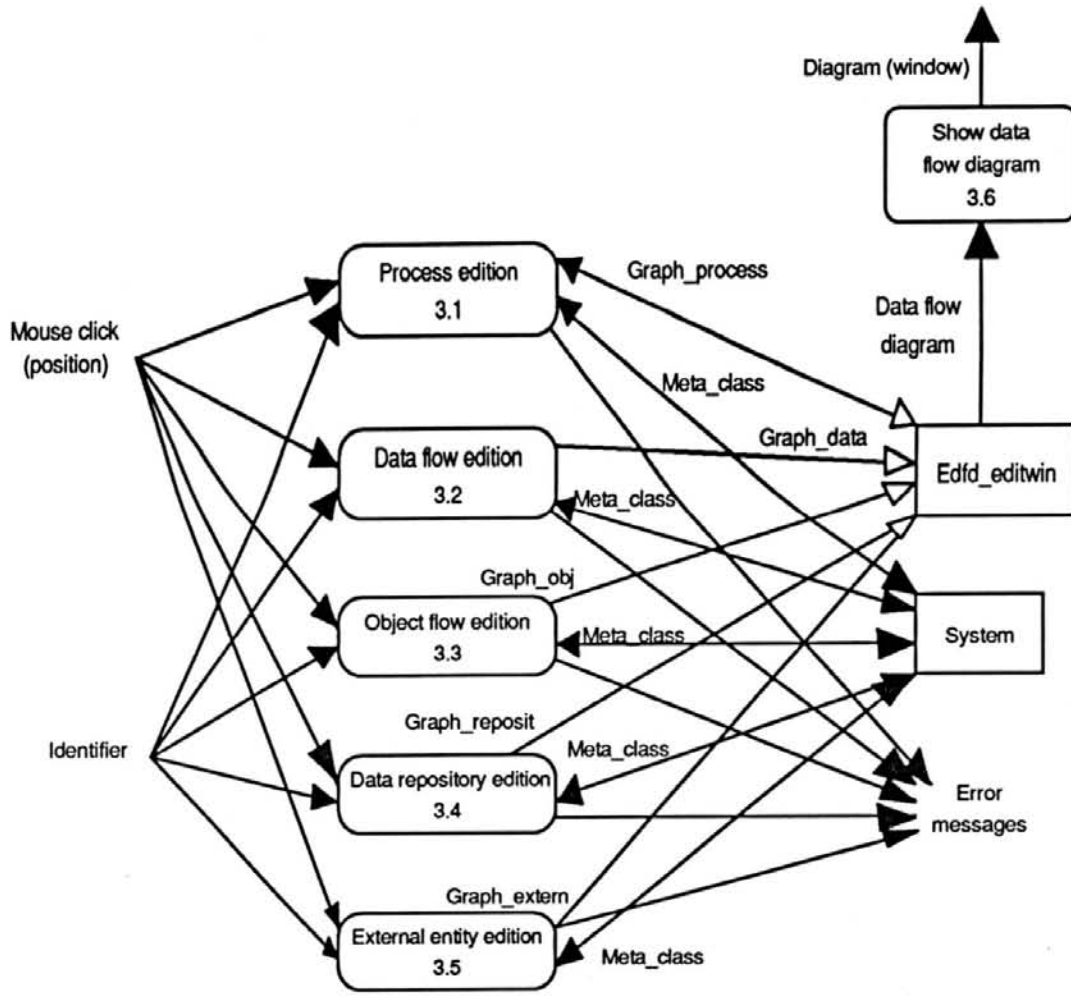

NÍVEL 0



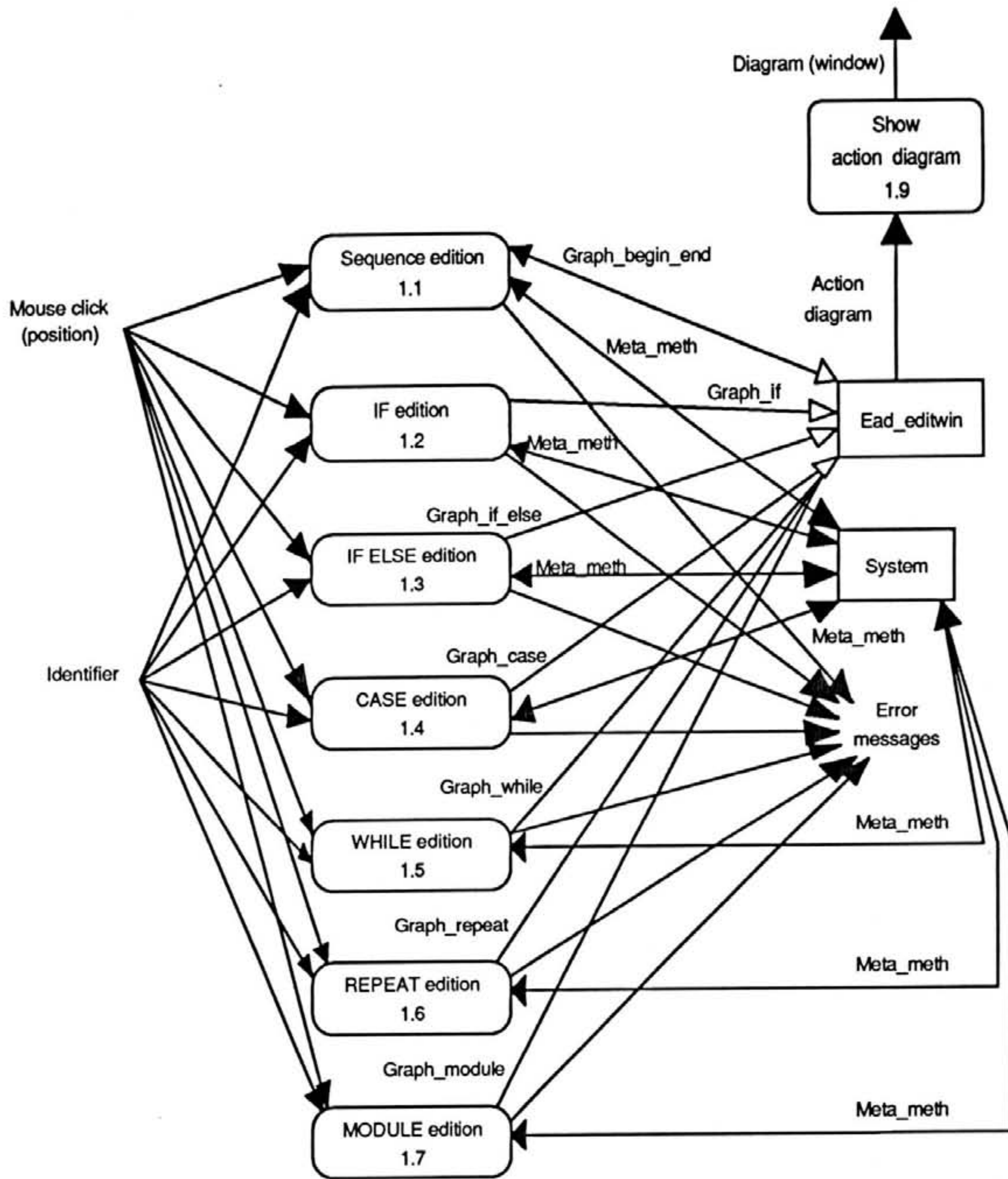

NÍVEL 1 ( Processo 1 - Edition of class/object diagrams)



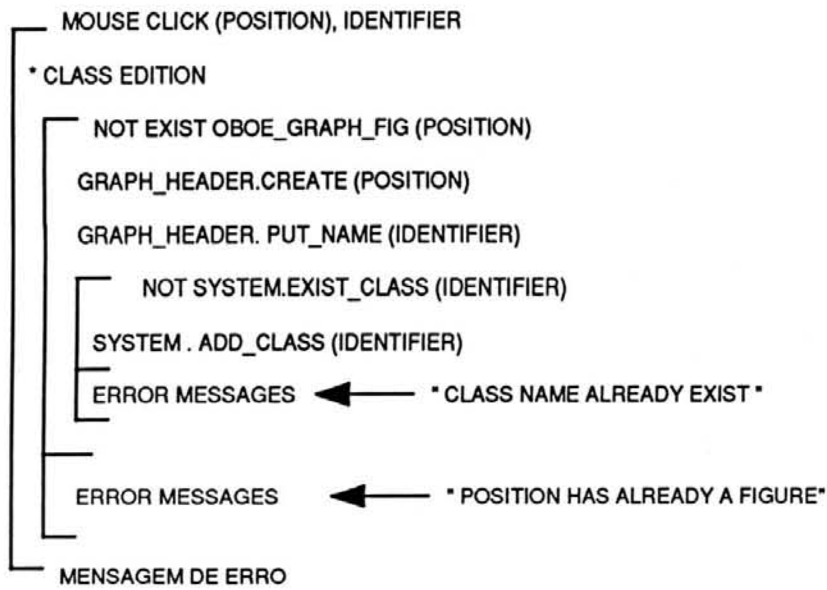

NÍVEL 1 ( Processo 2 - Edition of statecharts )

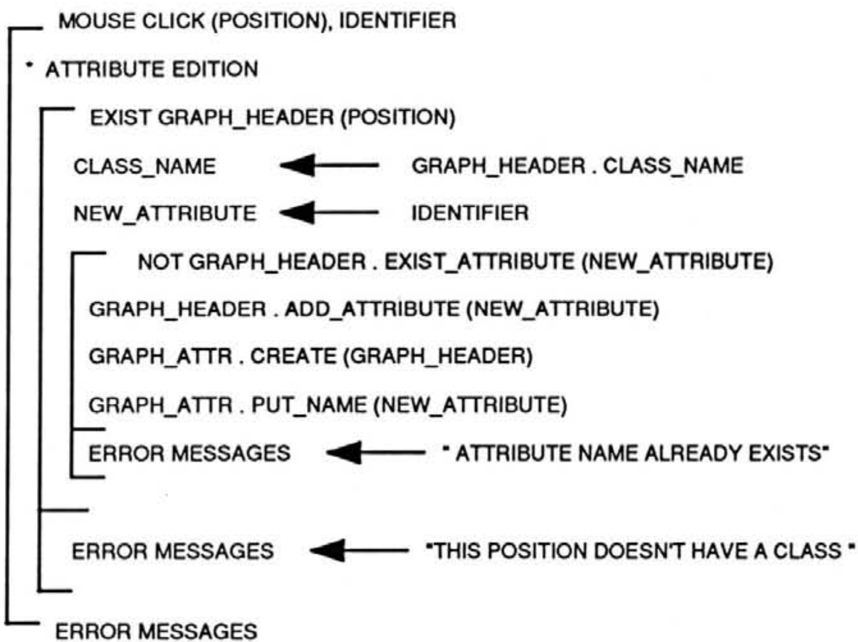


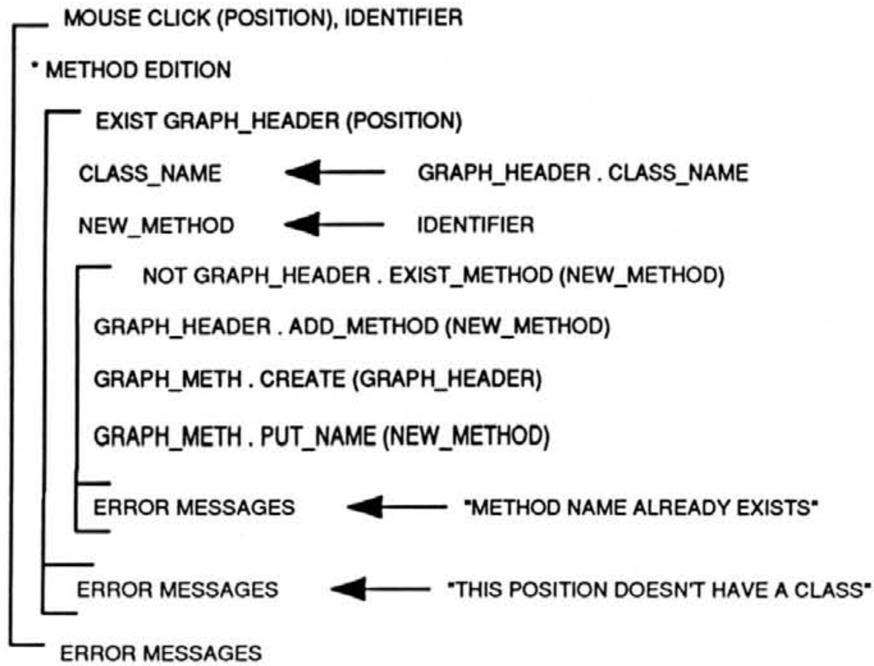

NÍVEL 1 ( Processo 3 - Edition of data flow diagrams)

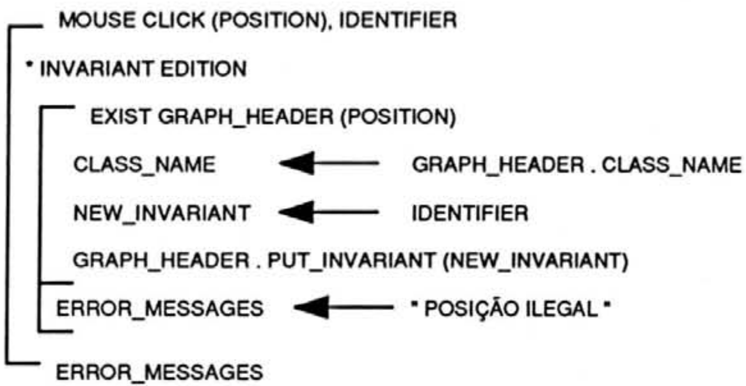



NÍVEL 1 ( Processo 4 - Edition of action diagrams)


MOUSE CLICK (POSITION), IDENTIFIER
* CLASS EDITION
NOT EXIST OBOE_GRAPH_FIG (POSITION)
GRAPH_HEADER.CREATE (POSITION)
GRAPH_HEADER. PUT_NAME (IDENTIFIER)
NOT SYSTEM.EXIST_CLASS (IDENTIFIER)
SYSTEM . ADD_CLASS (IDENTIFIER)
ERROR MESSAGES
" CLASS NAME ALREADY EXIST "
ERROR MESSAGES
" POSITION HAS ALREADY A FIGURE"
MENSAGEM DE ERRO


MOUSE CLICK (POSITION), IDENTIFIER
* ATTRIBUTE EDITION
EXIST GRAPH_HEADER (POSITION)
CLASS_NAME
GRAPH_HEADER . CLASS_NAME
NEW_ATTRIBUTE
IDENTIFIER
NOT GRAPH_HEADER . EXIST_ATTRIBUTE (NEW_ATTRIBUTE)
GRAPH_HEADER . ADD_ATTRIBUTE (NEW_ATTRIBUTE)
GRAPH_ATTR . CREATE (GRAPH_HEADER)
GRAPH_ATTR . PUT_NAME (NEW_ATTRIBUTE)
ERROR MESSAGES
" ATTRIBUTE NAME ALREADY EXISTS"
ERROR MESSAGES
"THIS POSITION DOESN'T HAVE A CLASS "
ERROR MESSAGES


MOUSE CLICK (POSITION), IDENTIFIER
* METHOD EDITION
EXIST GRAPH_HEADER (POSITION)
CLASS_NAME
GRAPH_HEADER . CLASS_NAME
NEW_METHOD
IDENTIFIER
NOT GRAPH_HEADER . EXIST_METHOD (NEW_METHOD)
GRAPH_HEADER . ADD_METHOD (NEW_METHOD)
GRAPH_METH . CREATE (GRAPH_HEADER)
GRAPH_METH . PUT_NAME (NEW_METHOD)
ERROR MESSAGES
"METHOD NAME ALREADY EXISTS"
ERROR MESSAGES
"THIS POSITION DOESN'T HAVE A CLASS"
ERROR MESSAGES


MOUSE CLICK (POSITION), IDENTIFIER
* INVARIANT EDITION
EXIST GRAPH_HEADER (POSITION)
CLASS_NAME
GRAPH_HEADER . CLASS_NAME
NEW_INVARIANT
IDENTIFIER
GRAPH_HEADER . PUT_INVARIANT (NEW_INVARIANT)
ERROR_MESSAGES
" POSIÇÃO ILEGAL "
ERROR_MESSAGES

# BIBLIOGRAFIA

[ ALA 88 ] ALABISO, Bruno. Transformation of data flow analysis models to object-oriented design. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS (OOPSLA), Sept. 25-30, 1988, Oslo. Proceedings... New York: ACM, 1988. p. 335-353.

[ BEA 90 ] BEAR, Stephen; ALLEN, Phillip; COLEMAN, Derek; HAYES, Fiona. Graphical specification of object oriented systems. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS (OOPSLA), Oct. 21-25, 1990, Ottawa; EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING (ECOOP), Oct. 21-25, 1990, Ottawa. Proceedings... New York: ACM, 1990. p. 28-37.

[ BOO 91 ] BOOCH, Grady. Object oriented design: with applications. Redwood City: The Benjamin/Cumming,1991. 580 p.

[ CHE 76 ] CHEN, Peter Pin Shan. The entity-relationship model - toward a unified view of data. ACM transactions on Database Systems, Baltimore, v.1, n.1, p. 9-36, Mar. 1976.

[ COA 91 ] COAD, Peter; YOURDON, Edward. Oriented-oriented analysis. 2nd ed. Englewood Cliffs: Prentice-Hall, 1991. 233 p.

[ DEM 78 ] DEMARCO, Tom. Strutuctured analysis and system specification. New York: Yourdon Press, 1978. 353 p.

[ ESP 89 ] ESPERANÇA, Lúcia G. Um interpretador de gramática de atributos. Porto Alegre: CIC da UFRGS, 1989. 114 p. (Trabalho de diplomação)

[ FAV 89 ] FAVERO, Elói L. Um editor orientado por estrutura para linguagens diagramáticas. Porto Alegre: CPGCC da UFRGS, 1989. 202 p. (Dissertação de mestrado)

[ GAN 79 ] GANE, Chris; SARSON, Trish. Structured systems analysis: tools and techniques. Englewood Cliffs: Prentice-Hall, 1979. 241 p.

[ HAR 88 ] HAREL, David. On visual formalisms. Communications of the ACM, New York, v.31, n. 5, p. 514-530, May 1988.

[ HEN 90 ] HENDERSON-SELLERS, Brian; EDWARDS, Julian M. The object-oriented systems life cycle. Communications of the ACM, New York, v. 33, n. 9, p. 142-159, Sept. 1990.

[ HEU 91 ] HEUSER, Carlos A. Modelagem conceitual de sistemas: redes de Petri. In: ESCOLA BRASILEIRO-ARGENTINA DE INFORMÁTICA (EBAI), 5., 1991, Nova Friburgo. Nova Friburgo: EBAI, 1991. 150 p.

[ INT 91 ] INTERACTIVE, Software Engineering Inc. Eiffel: the language, version 2.2. Goleta: Interactive, 1991. 278 p.

[ JAC 83 ] JACKSON, Michael A. System development. Englewood Cliffs: Yourdon-Press, 1983.

[ MCC 89 ] MCCLURE, Carma. CASE is software automation. Englewood Cliffs: Prentice-Hall, 1989. 290 p.

[ MAR 87 ] MARTIN, James. Recommended diagramming standards for analysts and programmers: a basis for automation. Englewood Cliffs: Prentice-Hall, 1987. 325 p.

[ MEL 89 ] MELO, Walcélio L. M. Proposta de um editor diagramático generalizado. Porto Alegre: CPGCC da UFRGS, 1989. 256p. (Dissertação de mestrado)

[ MEY 88 ] MEYER, Bertrand. Object-oriented software construction. Hemel Hempstead: Prentice-Hall, 1988. 534 p.

[ MRA 89 ] MRACK, Flávio R. Protótipo de um dicionário de dados para um editor diagramático generalizado. Porto Alegre: CIC da UFRGS, 1989. 110 p. (Trabalho de diplomação)

[ NAS 73 ] NASSI, I.; SHNEIDERMAN, Ben. Flowcharting techniques for structured programming. ACM SIGPLAN Notices, v. 8, n. 8, p. 12-26, Aug. 1973.

[ NER 91] NERSON, Jean-Marc. Analysis and design methodologies. In: INTERNATIONAL CONFERENCE & EXHIBITION, 5., July 29 - Aug. 1, 1991, Santa Barbara; INTERNATIONAL CONFERENCE ON TECHNOLOGY OF OBJECT-ORIENTED LANGUAGES AND SYSTEMS (TOOLS), July 29 - Aug. 1, 1991, Santa Barbara. Tutorial Notes... Santa Barbara: University of California, 1991.

[ PAG 88 ] PAGE-JONES, Meilir. Projeto estruturado de sistemas. São Paulo: McGraw-Hill, 1988. 396 p.

[ PAG 90 ] PAGE-JONES, Meilir; CONSTANTINE, Larry L.; WEISS, Steven. Modeling object-oriented systems: the uniform object notation. Computer language, v. 7, n. 10, p. 69-87, Oct. 1990.

[ PRI 84 ] PRICE, Roberto T. A prototype syntax driven language editor. Brighton, University of Sussex, 1984. 246 p. (Tese de doutorado).

[ REN 82 ] RENTSCH, T. Object-Oriented Programming. SIGPLAN Notices. v.17, n. 9, p. 51-58, Sept. 1982.

[ RUM 91 ] RUMBAUGH, James; BLAHA, Michael; PREMERLANI, William; EDDY, Frederick; LORENSEN, William. Object-oriented modeling and design. Englewood Cliffs: Prentice-Hall, 1991. 500 p.

[ SHL 88 ] SHLAER, Sally; MELLOR, Stephen J. Object-oriented systems analysis: modeling the world in data. Englewood Cliffs: Yourdon Press Computing Series, 1988. 144 p.

[ SIL 89 ] SILVA, Mônica S. Um formatador de diagramas. Porto Alegre: CIC da UFRGS, 1989. 75 p. (Trabalho de diplomação)

[ STR 86 ] STROUSTRUP, Bjarne. The C++ programming language. Reading: Addison-Wesley, 1986.

[ STR 88 ] STROUSTRUP, Bjarne. What is object-oriented programming?. IEEE Software. v. 5, n. 3, p. 10-20, May 1988.

[ TES 81 ] TESLER, L. The Smalltalk environment. Byte. v. 6, n. 8, p. 90-147. Aug. 1981.

[ WAR 86 ] WARD, Paul T.; MELLOR, Stephen J. Structured development of real-time systems. Englewood Cliffs: Yourdon Press, 1986.

[ YAM 91 ] YAMAGUTI, Marcelo H. Geração automatizada de gramática de atributos para técnicas diagramáticas. Porto Alegre: CPGCC da UFRGS, 1991. 59 p. (Trabalho individual)

[ YOU 89 ] YOURDON, Edward. Modern structured Analysis. Englewood Cliffs: Prentice-Hall, 1989. 836 p.

*Técnicas Diagramáticas para Desenvolvimento de Software Orientado a Objetos.*

Dissertação apresentada aos Senhores:

Profa. Carla Maria Dal Sasso Freitas

Prof. Dr. Carlos Alberto Heuser

Prof. Dr. Roberto Tom Price

Prof. Dr. Rogério Drummond (UNICAMP-SP)
BURNIER PESSÔA DE MELLO FILHO

Vista e permitida a impressão.
Porto Alegre, 17/ 11/ 93 .

Prof. Dr. Roberto Tom Price,
Orientador.

Prof. Dr. Ricardo A. da L. Reis,
Coordenador do Curso de Pós-Graduação
em Ciência da Computação.