Rêmulo Maia Alves

JSD-OO: Uma Adaptação do Método JSD à Orientação a Objetos

Dissertação apresentada ao Instituto de Ciências Exatas da UFMG como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte - MG
1993

"Quando voce não sabe onde quer chegar, todos os caminhos levam a lugares errados."

DEDICO

Aos meus pais, Silvia e Murilo, exemplo de honestidade, por todo esforço, carinho, estímulo e dedicação desprendidos ao longo de minha formação pessoal e profissional.

À minha esposa Maria, e meu filho Gustavo, pelo carinho dedicado e renúncia às horas de convívio.

Aos meus irmãos, pelo carinho e incentivo constante.

AGRADECIMENTOS

À Deus, pela graça de manter-me vivo e saudável durante todo período de duração do curso.

Ao Professor Doutor José Luís Braga pela dedicação e amizade com que sempre me orientou, pela sugestão do tema e pela confiança em mim depositada.

Ao Professor Doutor Alberto Henrique F. Laender pelas opiniões e conselhos acerca do material pesquisado, bem como pelos constantes ensinamentos transmitidos.

Ao Professor Doutor Crediné Silva de Menezes pelas opiniões precisas, seriedade e firmeza profissional.

Aos demais professores do DCC pela competência com que souberam ministrar os seus ensinamentos.

Ao sobrinho e amigo Rafael Matos Alves pela inestimável cooperação no desenvolvimento do protótipo da ferramenta CASE.

Ao meu irmão Murilo M. Alves pelas observações e comentários sobre o trabalho.

Aos amigos José Américo T. Messias e Ricardo S. Ferreira pelo grande auxílio prestados durante a elaboração da dissertação.

Aos amigos Vladimir, Mônica e Joyce pelo estímulo, solidariedade e convívio.

Às funcionárias Maristela, Emília e Rosângela o meu agradecimento pela dedicação, atenção, paciência e apoio durante o curso.

À Coordenação de Apoio de Pessoal de Nível Superior (Capes) através da Coordenadoria de Pós Graduação (CPG) da ESAL, pela bolsa de estudos concedida.

Aos colegas do Departamento de Ciências Exatas (DEX) da ESAL pela liberação para cursar o mestrado.

À todos que colaboraram, de alguma forma, na realização deste trabalho, o meu muito

obrigado.

FOLHA DE APROVAÇÃO

"JSD-OO: UMA ADAPTAÇÃO DO MÉTODO JSD À ORIENTAÇÃO A OBJETOS"

RÊMULO MAIA ALVES

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. JOSÉ LUÍS BRAGA - Orientador
Departamento de Informática – UFV

Prof. Alberto Henrique Frade Laender - Co-orientador
Departamento de Ciência da Cmputação - ICEx - UFMG

PROF. CREDINÉ SILVA DE MENEZES
Departamento de Informática - UFES

Belo Horizonte, 01 de outubro de 1993.

Resumo

Este trabalho faz considerações sobre o método JSD (Jackson System Development) assim como para os métodos orientados a objetos (OO), em especial o método OOD de Booch.

Neste estudo são feitas comparações entre os métodos JSD e OOD, enfatizando conceitos e técnicas usadas pelo paradigma de orientação a objetos presentes na teoria do método JSD e identificando elementos e características ausentes para que ele possa ser considerado um método orientado a objetos.

Baseado neste estudo, são propostas algumas técnicas novas bem como umas poucas extensões às existentes, transformando o JSD em um método orientado a objetos, o JSD-OO.

Essa tarefa não ficaria completa sem um ferramenta de auxílio ao desenvolvimento de sistemas (CASE) que suportasse o método. Assim esse trabalho teve por objetivo o desenvolvimento de um protótipo de ferramenta chamado JSD-OO TOOL. Seu objetivo principal é auxiliar o projetista a construir um modelo de sistema baseado nos conceitos do método JSD-OO, usando um dicionário de dados, diagramas e validando funções.

Em adição ao desenvolvimento do protótipo este trabalho também apresenta uma visão geral do uso do JSD-OO TOOL.

Abstract

This work is intended to present considerations about the Jackson System Development (JSD) method as well as some object-oriented (OO) methods, in special, Booch's OO-Design.

In this study, we compare the JSD and OOD methods, emphasizing the concepts and techniques used by the OO approach which are also present in Jackson's theory and identifying the elements and features which lack to the JSD's so that it can be considered a fully OO-method.

Based on this analysis, we propose same new techiques as well as (a few) extensions to the existing ones in order to create a real JSD-OO.

The task wouldn't be complete without a CASE tool to support the method. So we developed a prototype of such a tool, called JSD-OO TOOL. Its main purpose is to help the analyst/designer to build a system model based on JSD-OO concepts, using its dictionary, diagramming and validating functions.

In addition to the JSD-OO foundations, this paper also encompasses an overview of the JSD-OO TOOL.

Sumário

Lista de Figuras

Capítulo 1

Introdução

1.1 Considerações Iniciais

A influência do paradigma de **Orientação a Objetos** sobre os métodos de Análise e Projeto de Sistemas faz-se sentir cada vez com mais intensidade. A economia de conceitos, a simplicidade e aderência das primitivas de modelagem à forma como nós percebemos o mundo real, tornam esse paradigma uma opção, quase que obrigatória, quando se considera a evolução desses métodos.

De modo geral, as mudanças se fazem sentir por duas formas principais:

- **evolutivas**, em que métodos existentes e de larga utilização são adaptados ao novo paradigma. Essa forma permite que a comunidade usuária do método em questão absorva os novos conceitos linearmente, evitando-se saltos que podem até inviabilizar a absorção das novas idéias.

- **revolucionárias**, em que novos métodos são propostos, provocando a readaptação radical dos usuários de métodos já consagrados, caso optem pela utilização do novo paradigma. Essa forma é mais traumática, provocando algumas vezes mudanças bruscas e de difícil absorção pelos usuários. Em termos de custos globais, essa forma tende a ser mais cara que as mudanças evolutivas.

De maneira similar ao ocorrido na década de 70 com os Métodos Estruturados para análise e projeto de sistemas, estabelecidos com base no paradigma **procedimental** associado a técnicas de **decomposição funcional** [Yo79], nascidas a partir da proposição e utilização em larga escala de técnicas de programação estruturada, o paradigma de **Orientação a Objetos** para uso em análise e projeto de sistemas também tem evoluído a partir do uso cada vez maior das **linguagens orientadas a objetos** [Ko90]. A partir da experiência de uso e divulgação dessas linguagens, e do reconhecimento das facilidades proporcionadas pelas mesmas e pelos conceitos em que elas se baseiam, novos métodos de projeto baseados no paradigma de orientação a objetos têm sido propostos, sendo que essa evolução tende a chegar aos métodos de análise, completando assim o ciclo de síntese de novos modelos.

O método **Jackson System Development - JSD** [Ja83] para projeto de sistemas, proposto e divulgado por Michael A. Jackson na década de 80, baseia-se em um conjunto de princípios e primitivas que são de fácil entendimento. A notação adotada para a construção de esquemas baseados no método é também simples, concisa e livre de ambigüidades, derivada do método de programação do mesmo autor [Ja75], a partir do qual o JSD evoluiu.

No JSD, o mundo real é modelado a partir de objetos identificados, e interações entre eles, com a imposição da noção do seqüenciamento de ações sofridas e executadas pelos objetos. Essa similaridade de conceitos, detectada por vários autores [Bo86, Ve91], adeptos do paradigma de orientação por objetos, tornam o JSD alvo de discussões e dissecamento, na tentativa de torná-lo utilizável dentro dos preceitos do novo paradigma [Ve91]. Esses estudos encontram-se ainda em fase embrionária, e sua disseminação sob essa ótica ainda não foi feita em larga escala pela comunidade.

Este trabalho adota a premissa **evolutiva** para a introdução do paradigma de orientação a objetos. Partindo do método JSD, e do estabelecimento dos requisitos dos métodos orientados a objetos [Ch92], faz-se uma análise de aderência entre os dois. A partir dessa análise, reformulações e adições são propostas ao método JSD através da apresentação do método JSD-OO, adequando o JSD ao paradigma de orientação a objetos. O objetivo não é a formulação de um novo método, mas tão somente proporcionar extensões aos conceitos já utilizados no JSD, permitindo aos seus usuários uma evolução incremental. A partir das idéias apresentadas para adequação do novo método construímos um protótipo de ferramenta CASE, o JSD-OO Tool, como meio de auxiliar o projetista no desenvolvimento de sistemas utilizando esse método.

1.2 Organização da Dissertação

No Capítulo 2 – *Métodos Básicos* – são apresentados os métodos JSD e OOD de uma forma resumida mostrando os passos necessários para que faça o desenvolvimento de sistemas utilizando esses métodos.

No Capítulo 3 – *Uma Comparação entre os Métodos JSD e OOD* – são feitas comparações entre conceitos do paradigma de *orientação a objetos* com os do modelo proposto pelo método JSD. Também são feitas comparações entre a notação gráfica dos dois métodos.

No Capítulo 4 –*Método JSD-OO* – apresentamos uma extensão do método JSD, aproveitando conceitos que se adaptam ao paradigma de orientação a objetos estendendo-o naquilo que lhe falta para se tornar um método de análise e projeto segundo esse paradigma. Ao final apresentamos um estudo de caso usando os novos conceitos do métdo JSD-OO.

No Capítulo 5 – *Ambiente de Apoio ao Uso do Método JSD-OO* – é apresentado um resumo da utilização de ferramentas computacionais para auxílio no desenvolvimento de sistemas e mostramos o protótipo de uma ferramenta (**JSD-OO Tool**) para modelagem de sistemas utilizando o método JSD-OO.

No Capítulo 6 – *Conclusões* – fazemos algumas considerações a respeito do desenvolvimento deste trabalho, bem como damos uma visão de futuras pesquisas que possam surgir a partir dele.

Capítulo 2

Métodos Básicos

A idéia principal deste trabalho está baseada em conceitos e estruturas de dois métodos de projeto de sistemas: o JSD e o OOD. São eles, portanto, considerados como métodos básicos nos quais irá se fundamentar toda a teoria para a apresentação do método JSD-OO. A seguir será mostrado um resumo dos dois métodos discutindo-se os seus pontos principais.

2.1 Método JSD

Este resumo é baseado na visão de alguns autores que escreveram sobre o método JSD. Não é intenção desta seção esgotar o assunto, mesmo porque não é esse o objetivo a ser alcançado. O método é apresentado em seus elementos essenciais como forma de mostrar que ele pode vir a ser utilizado na especificação de sistemas com base nos conceitos do paradigma de orientação a objetos, auxiliando a sedimentação das idéias apresentadas a fim de se alcançar o objetivo proposto.

2.1.1 Resumo do método

O JSD é um método para desenvolvimento de sistemas que tem recebido bastante atenção na literatura técnica especializada por suas características únicas [Ro89].

Em [[Za84]*citadopor*[Ro89]], Zave destaca o JSD como o método operacional em estágio mais avançado de formalização e utilização prática, pois ele tem procedimentos e diretrizes claras para derivar a especificação operacional. O método fundamenta-se na criação de um modelo da realidade, baseado em eventos do mundo real, isto é, o modelo simula a realidade. O método JSD se aplica a sistemas em que os objetos de interesse realizam ou sofrem operações sujeitas a restrições no tempo [Me87].

A especificação nascida pela utilização do método JSD pode ser, em tese, diretamente implementada para ser executada pelo processador através de uma liguagem qualquer de programação [Ca86]. Entretanto, uma grande quantidade de processos, a frequência de vezes com que um processo é executado e o tempo de espera pela chegada de mensagens, fazem com que isso seja, na maioria das vezes, inviável na realidade [Me87].

Em [Ja78], Jackson propõe um sistema baseado em um modelo da realidade à qual pertence e que funções sejam superpostas a esse modelo. A forma do modelo proposto é o de uma rede de processos que se comunicam por seqüências de dados. Considera-se também que o modelo não pode ser eficientemente executado num único processador, sem que haja transformações nas especificações.

Em [Ja81], Jackson descreve quatro princípios em que o JSD se baseia:

1. O método decompõe a atividade de desenvolvimento em tarefas distintas e ordenadas, oferecendo para cada tarefa uma ferramenta adequada à sua execução e critérios para determinar se foi completada corretamente;

2. O projetista do sistema deve começar considerando *a realidade que o sistema vai modelar*, ao invés das *funções que o sistema vai realizar*;

3. Uma realidade ativa só pode ser modelada por um modelo ativo, assim como um modelo inerte pode modelar apenas uma realidade inerte;

4. O projeto deve *preceder*, e não *ser confundido* com a implementação.

Em [Ja83], Jackson divide o processo de desenvolvimento JSD em 6 etapas ou fases: Fase I – das escolhas das entidades e das ações, Fase II – de estruturação cronológica das ações, Fase III – de construção do modelo inicial, Fase IV – de adição das funções, Fase V – de temporização e Fase VI – da implementação do sistema.

Fase I *Escolha das Entidades e Ações*: no método JSD, o Sistema de Informação a ser desenvolvido deve ser estudado visando à percepção dos elementos centrais para a modelagem [Me87], que são as *entidades* e as *ações* de interesse no ambiente real do problema. A seleção das *entidades* e das *ações* delimita o escopo do sistema. Com isso fica limitado o conjunto de consultas que o sistema pode responder, tanto para objetos do mundo real quanto para ações a respeito desses objetos. Logo o mundo real é descrito em termos das *entidades*, das *ações* que elas requerem ou sofrem, e da ordenação destas *ações* no tempo [Ja83].

Uma *entidade* pode representar, segundo [Me87]; pessoas, coisas e organizações.

As *entidades* são mais bem percebidas pelas *ações* que elas executam ou sofrem no mundo real. Uma *entidade* em JSD jamais será uma coisa inerte, sem um padrão de comportamento definido ao longo do tempo.

Ações[1] compõem as *entidades* e são consideradas os blocos fundamentais de construção do modelo JSD [Su88].

O sistema deve manter um registro da situação de cada *entidade* e de como ela se altera a cada *ação* executada. Esse registro é mantido por diversos *atributos* relacionados à *entidade* [Ms89].

Como resultado desta fase o método deve produzir:

- Uma lista de *entidades* e *ações* (LEA);

[1] **Ações** no modelo JSD podem ser consideradas como **eventos**, isto é, ações são eventos que se relacionam com entidades e que afetam o ambiente do mundo real mudando a situação de cada entidade.

- Um Dicionário das *ações* com seus atributos relacionados.

O conceito de *entidade* e de *ação* é relativo e depende dos propósitos do sistema que será construído, e do ponto de vista do projetista sobre o mundo real a ser modelado.

As principais características de uma *entidade* são [[Ja83]*citadopor*[Me87]]:

1. Uma *entidade* tem que sofrer ou realizar ações em uma ordem significativa no tempo. Idade, número de créditos e carga horária, por exemplo, são difíceis de serem concebidos como entidades;

2. Uma *entidade* deve existir no mundo real, fora do sistema, e não meramente ser uma parte ou produto do sistema. Ex.: "relação de turmas com menos de 15 alunos", não se caracteriza como entidade;

3. Uma *entidade* deve ser capaz de ser considerada como um indivíduo e, se houver mais que uma do mesmo tipo, devemos ser capazes de identificá-las univocamente. Ex.: "alunos do curso de engenharia" só pode ser considerada como entidade se não estamos interessados em cada aluno do curso de engenharia individualmente e sim no todo.

Menezes em [Me87] enumera as principais características de uma *ação*:

1. Uma *ação* é algo que ocorre em algum momento e nunca algo que se estende ao longo do tempo. Ex.: estar matriculado na disciplina "X" é um estado e não uma ação. Matricular-se em uma disciplina é uma ação da entidade aluno;

2. Uma *ação* deve existir no mundo real fora do sistema que iremos construir, e não pode ser meramente uma *ação* do sistema em si. Ex.: "determine os alunos com créditos completos" é uma ação do sistema;

3. Uma *ação* é sempre atômica, não podendo portanto ser dividida em sub-ações. Ex.: "matricular-se em uma disciplina" e "cancelar a matricula" são ações da entidade aluno, então não podemos ter uma ação "transfere a matrícula de uma disciplina para outra" como uma ação da entidade aluno.

Em [Ja83], Jackson descreve uma outra idéia sobre o desenvolvimento de sistemas pelo método JSD, o conceito de *Tipo-Entidade*. Abaixo são descritas as suas características principais:

- Quando se faz uma seleção de *entidades* dentro do domínio da aplicação, na verdade, o que está sendo feito são duas diferentes espécies de escolha simultâneamente: (1) "coisas individuais" e/ou "pessoas" que podem compor a descrição abstrata do mundo, e (2) como eles podem ser classificados;

- Um *tipo-entidade* é portanto o conjunto de todas as *entidades* da realidade que possuem a mesma *estrutura* e sofrem ou praticam as mesmas *ações* (comportamento);

- Em JSD, a escolha dos *tipos-entidade* é limitada a duas considerações:

1. um *tipo-entidade* é especificado em termos das *ações* possíveis requeridas ou sofridas pelas instâncias do tipo, e pela ordenação destas *ações* no tempo;
2. um tipo-entidade é especificado por todo o tempo de vida de uma *entidade*, isto é, por toda a sua existência dentro dos limites do modelo.

- Uma *entidade* pode ingressar no sistema, abandonar o sistema, mas nunca trocar de tipo-entidade;
- Uma *entidade* pode desempenhar dois papéis distintos (pertencer a dois *tipos-entidade* diferentes) em um mesmo modelo da realidade. Nesse caso a modelagem deve:
  1. ou criar um *tipo-entidade* mais geral que possua os dois conjuntos de *ações*;
  2. ou deixar de fora do modelo o conhecimento de que uma instância de uma *entidade* pode representar dois papéis diferentes.

Fase II *Estruturação Cronológica das Ações*: em JSD a estruturação cronológica mostra a ordenação no tempo das *ações* que podem ocorrer em uma *entidade*[2]. A ordenação é dada pelo Diagrama de Estruturação Cronológica (DEC) através das estruturas de *Seqüência*, *Iteração* e *Seleção* de ações.

A estrutura do DEC é uma árvore onde os descendentes de cada nodo só podem ser uma das seguintes estruturas: seqüência, iteração ou seleção.

Os ícones utilizados para a representação de um DEC são mostrados na Figura 2.1.

Para cada *entidade* deve existir um Diagrama de Estruturação Cronológica, às vezes chamado apenas de Diagrama de Estrutura (DE).

Uma descrição instantânea da realidade modelada pode ser dada através das *entidades*, das *ações*, da ordem em que essas *ações* são executadas, e por último pelo estado em que se encontram os *atributos* das *entidades* (vetor de estados). Toda *entidade* é modelada através de um *processo sequencial* [Me87].

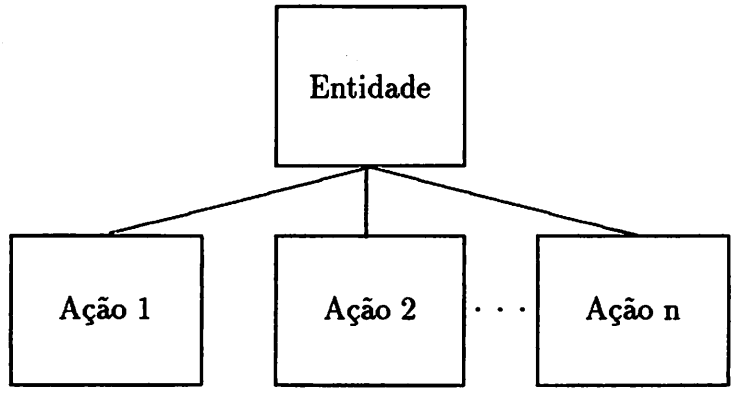
Fase III *Construção do Modelo Inicial*: nessa fase o modelo iniciado nas fases anteriores deve ser conectado com o mundo real. Para cada *entidade* deve ser criado um processo dentro do sistema e um processo no mundo real conectados, na maioria das vezes, através de uma cadeia de mensagem [Me87]. O objetivo da cadeia de mensagem é alimentar o sistema com os fatos ocorridos com a *entidade* correspondente do mundo real.

No método JSD existem duas formas de conexão de processos que são usadas tanto para conectar os processos de modelagem ao mundo real bem como para conectá-los a processos internos ao sistema. Essas conexões são:

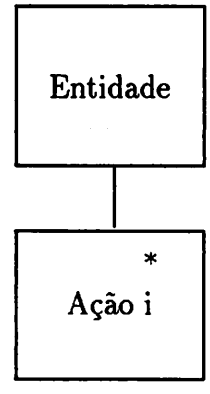
- *Conexão por Seqüência de Dados* (Data Stream), onde um processo escreve uma seqüência de dados, consistindo de um conjunto ordenado de mensagens ou registros, e o outro processo da rede de processos lê essa seqüência;

[2]Em [Me87] Menezes considera as Fases I e II como constituindo uma única fase por ele chamada de Fase de Identificação de Entidades e Ações.

Sequência de Ações



Iteração de uma Ação



Seleção de uma Ação

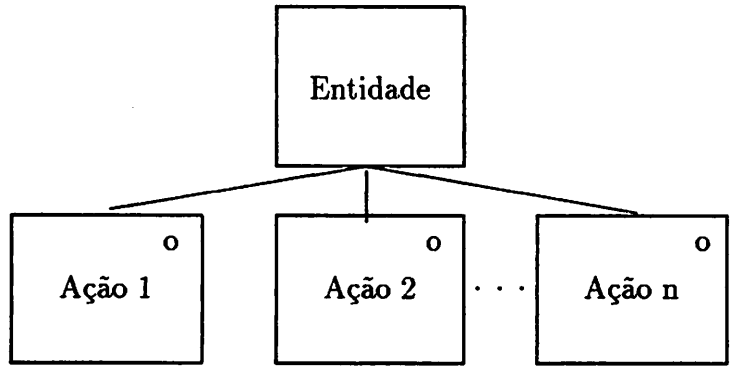


Figura 2.1: Ícones para Representação das Estruturas JSD

- *Conexão por Inspeção do Vetor de Estados* (State Vector), onde um processo lê um conjunto de variáveis locais (vetor de estados) pertencentes, internamente, a outro processo da rede de processos.

As características da *conexão por inspeção de vetor de estados* são:

1. A iniciativa da comunicação é de responsabilidade do processo que deseja a informação (*receptor*);
2. Os valores assumidos pelo vetor de estado do *emissor* dependem inteiramente da estrutura deste;
3. A cada operação do *emissor* seu vetor de estado é modificado sem nenhum compromisso com o *receptor*, assim nem todos os estados atingidos pelo *emissor* são obrigatoriamente percebidos pelo *receptor*;
4. A operação de inspeção não bloqueia nenhum dos processos que participam da conexão.

Em [Ja83], Jackson define duas maneiras para entrada de dados no modelo:

Múltiplas entradas para um processo (Multiple Input): um processo pode ter múltipla entradas e múltiplas saídas, as quais podem ter diferentes instantes de chegada ou partida, respectivamente. As mensagens das seqüências de dados devem ser observadas de acordo com regras determinadas na especificação, mantendo uma ordem de leitura que melhor satisfaça ao sistema em desenvolvimento.

O tratamento de múltiplas entradas por um único processo é chamado de *Intercalação Fixa* (Fixed Merge) quando as leituras das seqüências de dados ocorre numa ordem predeterminada, ou então de *Intercalação por Data* (Data Merge), quando duas seqüências são lidas de acordo com uma certa *data*.

Intercalação Natural ou Merge Rústico (Rough Merge): este tipo de intercalação ocorre quando um processo recebe múltiplas entradas e as lê conforme a ordem de chegada das mensagens, independentemente da ordem na qual elas foram escritas ou da velocidade do canal de comunicação. O processo que absorve as seqüências recebe as mensagens como se elas chegassem por uma única entrada. Graficamente, no DES, a intercalação natural é representada por duas ou mais seqüências de dados cujas setas se unem no lado do processo que as absorve.

Nesta etapa inicia-se a construção do Diagrama de Especificação do Sistema (DES), contendo os processos de modelagem e estabelecendo suas conexões com os processos do mundo real estabelecidas.

Graficamente, no DES, os processos são representados por retângulos, as conexões por mensagens através de círculos, e as conexões por inspeção do vetor de estados através de losangos. As flechas indicam o sentido da conexão (Figura 2.2).

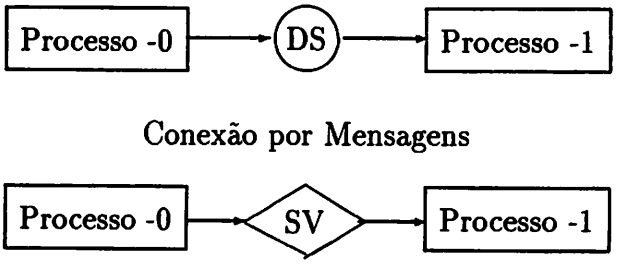


Figura 2.2: Ícones para Representação do Diagrama de Especificação de Sistemas

No modelo inicial, o DES é composto por processos de dois níveis: nível zero e nível maior que zero. Os processos de nível zero recebem o nome da entidade seguido do sufixo "0" e representam o objeto do mundo real que fornece as entradas ao sistema. Os processos de nível maior que zero são os correspondentes aos processos de modelagem que simulam o comportamento da entidade dentro do computador, e são nomeados com o próprio nome da entidade, seguido do sufixo "< *no.maiorquezero* >".

O projetista, quando chega a esta fase, deve especificar as estruturas dos processos escrevendo o que Jackson em [Ja83] denominou *texto estruturado.* O *texto estruturado* é essencialmente uma forma textual do DEC, semelhante ao texto de um procedimento escrito em uma linguagem simbólica (pseudocódigo). Os comandos e expressões do texto estruturado se assemelham em muito com os comandos e expressões da linguagem PASCAL, não havendo entretanto nenhuma regra fixa para a sua escrita, já que o texto não será compilado ou executado por nenhum processador. As principais regras e comandos do texto estruturado estão definidas no Apêndice A - Texto Estruturado.

Como resultado desta fase o método deve fornecer:

- Revisão da lista de *entidades* e *ações*;
- Um Diagrama de Especificação de Sistemas;
- Diagramas de Estruturação Cronológica para processos extras;
- Texto Estruturado para todos os processos do sistema;

Fase IV *Adição de Funções / Rede de Processos*: nessa etapa o método JSD continua trabalhando com o DES, permitindo a inserção de novos processos de modelagem do sistema ao Modelo Inicial, assim como de todos os processos que geram as saídas produzidas pelo sistema, chamados em JSD de *Processos Funcionais.*

Funções são componentes do sistema em desenvolvimento, as quais combinadas com os processos do Modelo Inicial formam o sistema como um todo.

Em [Ca83], Cameron divide as funções em dois grandes grupos:

1 - Funções de Informação: Tem por objetivo fornecer aos usuários do sistema um acompanhamento dos eventos ocorridos no mundo real através da observação dos processos do modelo. Ex.: Informar ao cliente de um Banco todas as vezes que sua conta fica com saldo negativo. Uma Função de Informação pode ser subdividida ainda em:

a. **Função Embutida:** Quando a combinação dos eventos é simples e a computação da função é direta, as operações da função podem ser diretamente embutidas na estrutura dos processos do Modelo Inicial;

b. **Função Imposta:** Eventos podem ser solicitações do usuário do sistema (solicitação externa ao sistema). Nesse caso a função será um novo processo que inspecionará as variáveis locais do vetor de estados de algum processo.

2 - Funções Interativas ou de Automação: Tem por objetivo automatizar ações que a princípio poderiam ser realizadas no mundo real, isto é, ações geradas por processos internos ao modelo e que portanto não possuem conexão com o mundo real. Isso ocorre quando um processo funcional consulta o vetor de estados de um processo do modelo, e posteriormente a saída gerada será a entrada desse ou de outro processo do modelo [Ja83].

Em [Ca86], Cameron acrescenta ainda as *Funções de Manipulação de Erros* colocadas entre a realidade e o modelo. Sua principal função é colecionar as informações sobre as *ações* tanto quanto possível para que somente dados livres de erros sejam passados ao modelo.

Genericamente podemos dizer que cada função requerida corresponde à adição de um novo processo para realizar a função [Me87].

Como resultado desta fase o método JSD deve produzir:

- Uma reformulação da descrição da função com base no Modelo Inicial existente (DES revisado), mostrando os processos que realizam as funções internas e suas conexões;
- Uma revisão do Texto Estruturado de processos alterados para embutir ou apenas produzir acréscimos para as novas funções do sistema;
- Um Texto Estruturado e um DEC descrevendo as novas Funções;
- Uma descrição concisa dos caminhos de acesso aos vetores de estados definido por novas funções.

Fase V *Temporização dos Componentes do Sistema*: para modelar uma ação que seja executada a partir de um instante de tempo ou mesmo por um período de tempo, o JSD introduz o conceito de *Marcadores de Intervalo de Tempo*, que indicam a chegada, ao mundo real, desse instante no tempo.

Sistema e realidade caminham defasados, ou seja o sistema está quase sempre em atraso em relação à realidade.

Nesta fase, o projetista deve considerar todas as potenciais fontes de retardo e determinar com os usuários os limites aceitáveis para cada parte do sistema.

A documentação produzida nesta fase é bastante informal, em geral, constando de notas sobre tolerância para os vários componentes.

Fase VI *Implementação do Sistema*: durante as fases anteriores a maior preocupação era com a modelagem da realidade de maneira a atender, da melhor maneira possível, os requisitos da especificação. Não foi levado em consideração o fato de que na maioria das vezes existem mais processos do que processadores.

Nesta fase o método JSD apresenta um conjunto de técnicas as quais devem ser usadas sistematicamente para converter uma especificação em um sistema executável.

Os principais aspectos do método de Jackson para esta fase são [Me87]:

Transformações: são técnicas utilizadas com o objetivo de reduzir a quantidade de processadores e memória necessários para executar a especificação. Para se obter a redução dos processadores devemos fazer um compartilhamento deles levando-se em conta dois aspectos:

1. Reduzir o número de processadores para fazer com que *processos de um mesmo tipo* sejam executados em um número de processadores menor que o número de processos. A esta técnica damos o nome de *Separação do Vetor de Estados.* Isto se deve porque a única diferença entre um processo e outro do mesmo tipo é o estado de sua variáveis locais. A técnica consiste portanto em separar o vetor de estado do texto do processo e armazená-lo em um dispositivo de memória identificado pelo código do processo;
2. Reduzir o número de processadores, compartilhando seu tempo com dois ou mais processos de tipos diferentes (*Inversão de Programas*).

Escalonamento: processos escalonadores são adicionados ao projeto durante a fase de implementação para controlar o tempo de execução e a seqüência de chamada dentro de um sistema. Os processos são *invertidos* em relação ao processo escalonador, o que significa que o *escalonador* torna-se o programa principal e os processos do sistema suas subrotinas chamadas por ele.

Às vezes um processo do sistema pode tornar-se um programa escalonador, embutindo no texto do processo a tarefa do compartilhamento dos processos. Em geral isso só é conveniente nos casos em que:

- Apenas conexão por seqüência de mensagens é usada;
- Não há Merge Rústico no sistema;
- Quaisquer dois processos são conectados apenas por uma cadeia.

As primeiras quatro fases do ciclo de vida, desenvolvidas pelo método JSD, têm a ver com a criação de uma *Especificação de Requisitos do Sistema* e as duas últimas com a

Implementação dessa Especificação. Em [Ca86], Cameron afirma que as especificações em JSD consistem principalmente de uma rede distribuída de processos seqüenciais que se comunicam ou por troca de mensagens de leitura e escrita de seqüências de dados, ou por consulta aos seus vetores de estado. Desse modo, ele afirma que uma especificação JSD pode ser diretamente executável, pelo menos em princípio e algumas vezes também na prática.

Em [Ca86], Cameron considera o método JSD como sendo formado por 3 fases principais:

1. Fase de **MODELAGEM**: seleciona e define os processos modelados, correspondendo aos itens 1, 2 e 3 das 6 etapas do desenvolvimento JSD;
2. Fase de **REDE**: desenvolve o resto da especificação, correspondendo ao item 4 do desenvolvimento JSD;
3. Fase de **IMPLEMENTAÇÃO**: os processos e seus dados são encaixados nos processadores disponíveis e na memória, correspondendo aos itens 5 e 6 do desenvolvimento JSD.

2.2 Métodos de projeto orientados a objetos - MPOO

Os MPOO's são técnicas para o desenvolvimento de sistemas baseados na modelagem de objetos do mundo real e que formam o domínio do problema. Os sistemas nascidos a partir da utilização dessas técnicas permitem que o projetista tenha um melhor entendimento dos requisitos de construção e por isso um sistema de mais fácil manutenção, entre outras vantagens. Isso se deve em parte à transparência na passagem das fases de análise para a de projeto e dessa para a fase de implementação no ciclo de vida para o desenvolvimento de sistemas. Em todas essas fases, o elemento central é o *objeto* juntamente com os conceitos que o cercam, permitindo dessa forma que se utilizem as mesmas técnicas e a mesma notação durante todo o processo de desenvolvimento.

Em seguida será mostrado um resumo das principais partes que serão utilizadas como base de comparação com o JSD, de um desses métodos, o OOD (Object Oriented Design) desenvolvido e apresentado no livro de Grady Booch [Bo91].

2.2.1 Justificativa

Por quê escolher o OOD de Grady Booch como método de comparação com o JSD? Essa foi a principal pergunta que fizemos ao começar os estudos para o desenvolvimento desse trabalho. Achamos conveniente a escolha desse método em particular, primeiro devido à profundeza com que os aspectos que envolvem toda a teoria da Orientação a Objetos são tratados ao longo do livro [Bo91]. Aspectos como a complexidade intrínseca de sistemas, ou o papel da decomposição, da abstração e da hierarquia como elementos importantes para o fatoramento da complexidade dos sistemas, são tratados de uma forma bem natural à realidade que cerca o homem. Pelo que pudemos observar através das

referências bibliográficas e, pelo acompanhamento dos fatos cronológicos que marcaram o nascimento do paradigma de Orientação a Objetos até os dias atuais, Grady Booch pode ser considerado ao nosso ver, como um dos pioneiros na formalização e desenvolvimento de métodos orientados a objetos, sendo esse também considerado o principal motivo por sua obra ser citada como referência em trabalhos de outros autores. Entre os principais trabalhos de sua autoria sobre o assunto destacamos [Bo82], passando por [Bo86], até chegar em [Bo91], principal fonte de comparação com o método JSD neste trabalho.

O Segundo aspecto que motivou a escolha do OOD como parâmetro de comparação com o JSD foi devido às características de completeza que o método possui. Segundo David E. Monarchi *et alii* em [Mo92], os métodos de análise e projeto orientado a objetos (MAPOO) se classificam em três categorias de pesquisa, assim divididos:

1. métodos procedimentais para análise e projeto que não incluem notações diagramáticas para a sua representação,
2. MAPOO's que se preocupam apenas com a representação visual do projeto, não possuindo nenhuma técnica para desenvolvimento do sistema,
3. MAPOO's mais completos, que compreendem as duas categorias acima, e que portanto possuem os mecanismos para o desenvolvimento dos processos inerentes ao modelo, bem como uma notação diagramática para representar a especificação do resultado.

Segundo os autores dessa pesquisa, o método de projeto desenvolvido por Grady Booch, além de enquadrar-se na terceira categoria, é o método que mais características de desenvolvimento incorpora em sua estrutura, vinte e uma (21) no total, sendo considerado por isso o método mais completo dentre os pesquisados no trabalho.

2.2.2 Complexidade

A demanda sempre crescente por sistemas de programas, cada vez mais abrangentes e complexos, que muitas vezes excedem a capacidade intelectual humana, e a ausência de uma forma mais organizada e sistematizada no processo de seu desenvolvimento, resultaram na chamada "crise do software".

A complexidade pode muitas vezes tomar uma forma hierárquica, onde os elementos do domínio do problema podem de alguma forma cooperar entre si, formando estruturas mais complexas do modelo em desenvolvimento. Para que isso possa acontecer os métodos de desenvolvimento utilizam-se das chamadas *técnicas de decomposição e abstração.*

O método de projeto orientado a objetos apresentado se baseia na decomposição orientada a objetos, utilizando regras definidas e uma notação gráfica para a construção de sistemas complexos.

2.2.3 Modelo de Objetos

Os métodos de projeto estruturados surgiram de modo a orientar os projetistas de sistemas que construiam sistemas complexos a usarem *algoritmos* como primitivas para sua

construção. Similarmente, os métodos de projeto orientados a objetos têm evoluido para auxiliarem os projetistas a explorarem o poder expressivo das linguagens de programação baseadas em objetos e orientadas a objetos, usando *classes e objetos* como primitivas para construção de sistemas.

Não obstante haja uma tendência cada vez maior à adoção do novo modelo, ainda não existe uma padronização em muitos conceitos e na sua utilização no desenvolvimento de sistemas. De Champeaux e Faure em [Ch92], aproveitando a ausência de normas, fazem uma comparação entre os vários métodos que utilizam o modelo objeto, mostrando o que têm em comum e onde diferem.

A base conceitual da orientação a objetos é o *Modelo de Objetos* baseada em quatro elementos principais, sem os quais o modelo não pode ser considerado orientado a objetos [Bo91]:

Abstração: é o reconhecimento de similaridades entre certos objetos, situações ou processos no mundo real, e a decisão de concentrar-se nestas similaridades e ignorar as diferenças;

Encapsulamento: é a ocultação de todos os detalhes de um objeto, que não contribuem para suas características essenciais. Isso é conseguido por meio da utilização de funções e procedimentos que manipulam de uma forma única as estruturas de dados dos objetos;

Modularidade: é a propriedade que um sistema tem de ser decomposto em um conjunto de abstrações relacionados logicamente (módulos) minimizando ao mesmo tempo o relacionamento entre os módulos;

Hierarquia: a palavra hierarquia pode ser definida como "qualquer classificação que tenha como base as relações entre superiores e dependentes". No modelo de objetos a *hierarquia* também pode ser definida como uma categorização ou uma ordenação das abstrações.

Objetos e Classes

Quando se usam métodos orientados a objetos para projetar sistemas de informação, a base para a construção desses sistemas são as *classes* e os *objetos*, que propiciam a utilização dos quatro elementos principais, vistos acima.

Um objeto deve modelar alguma parte da realidade e é portanto alguma coisa que existe no tempo e no espaço. Sob a perspectiva do conhecimento humano um objeto pode ser:

- Uma coisa tangível ou visível. Ex.: um automóvel;
- Alguma coisa que deva ser compreendida intelectualmente. Ex.: um escritório de contabilidade;

- Alguma coisa em direção à qual pensamentos ou ações são direcionados. Ex.: um caixa de um banco.

Um objeto tem *estado, comportamento e identidade.* A estrutura e comportamento de objetos similares são definidos em suas classes comuns.

O *estado* de um objeto compreende todas as propriedades de um objeto – *características estáticas*, mais os valores correntes de cada uma destas propriedades – *características dinâmicas.*

O *comportamento* é a descrição de como um objeto age ou reage, em termos de mudanças no seu estado e da passagem de mensagens. Um objeto age sobre outros objetos utilizando-se das operações (o termo mensagem, método e operação são intercambiáveis) definidas nas classes desses objetos.

Identidade é a propriedade de um objeto, segundo a qual ele deve ser distinguido de maneira única dos outros objetos.

Os dois tipos mais comuns de operações realizados por um objeto são:

Modificadora: uma operação que altera o estado de um objeto. Ex.: um objeto *cliente* envia uma mensagem para alterar uma *requisição.* O objeto *requisição* altera o seu estado;

Seletora: uma operação que acessa o estado de um objeto mas não o altera. Ex.: um objeto *relatório* envia uma mensagem solicitando o saldo de um determinado objeto *cliente.* O estado do objeto *cliente* não é alterado.

A expressão *enviar uma mensagem* representa, no modelo de objetos, a forma convencional de estabelecer comunicação entre dois objetos quaisquer do sistema. As mensagens atuam como *avisos*, sinalizando a ocorrência de um evento [Ve91]. Elas são compostas de duas partes: uma que identifica a operação a ser executada por outro objeto e a outra (opcional) que leva as informações adicionais. Assim a expressão *enviar uma mensagem* representa a ação de "partindo de um objeto do sistema, chamar um procedimento definido em outro objeto qualquer, estabelecendo dessa forma uma comunicação entre os dois objetos".

Um objeto sózinho não tem interesse algum em termos de construção de sistemas. Os objetos contribuem para o comportamento do sistema, quando colaboram uns com os outros.

Existem dois tipos de relacionamento entre os objetos, de real interesse para o OOD que são:

1. Relacionamentos de Utilização: no OOD uma linha direta entre dois ícones representando objetos, mostra a existência de relacionamentos de utilização entre os dois, significando que mensagens passam por este caminho;

2. Relacionamentos de Composição: quando dois ou mais objetos de classes diferentes ou não são agrupados para formarem um outro objeto, nós estamos criando um relacionamento de composição entre eles, isto é, objetos de outras classes são encapsulados fazendo parte do estado do objeto.

Nos Relacionamentos de Utilização um objeto pode representar um dos três papéis a seguir:

- Ator: um objeto que *pode operar sobre outros* objetos mas que *nunca é operado por outros* objetos; às vezes também é utilizado com o nome de *objeto ativo*;
- Servidor: um objeto que *nunca opera sobre outros* objetos, é *somente operado por outros* objetos; conhecido às vezes também com o nome de *objeto passivo*;
- Agente: um objeto que pode tanto operar sobre outros objetos como pode ser também operado por outros objetos.

Quando um objeto passa uma mensagem para outro com o qual tenha um Relacionamento de Utilização, os dois objetos devem estar *sincronizados* de alguma maneira. Para objetos em uma aplicação completamente seqüencial a sincronização é feita por chamadas a subprogramas. Quando entretanto os objetos se envolvem com múltiplas situações ou estruturas de controle, isto é, com mais de um objeto ativo, a sincronização é mais sofisticada de modo a lidar com problemas de *exclusão mútua*[3] que ocorrem em sistemas concorrentes. Em relação à sincronização os objetos podem ser classificados como:

- Objetos Seqüenciais: um objeto passivo (servidor) cujo comportamento é ativado somente em presença de apenas uma linha de controle;
- Objetos de Bloqueio: um objeto passivo (servidor) cujo comportamento é ativado em presença de múltiplas linhas de controle;
- Objetos Concorrentes: um objeto ativo (ator) cujo comportamento é ativado em presença de múltiplas linhas de controle.

Outro conceito importante para o método OOD é o de *visibilidade*. Existem três modos pelos quais um objeto X pode ser feito visível a um objeto Y:

1. *Mesmo escopo léxico*, isto é, o objeto Y está dentro do objeto X, desta maneira X pode explicitamente referenciar Y;
2. *Por Parâmetro*, isto é, o objeto Y é passado como um parâmetro para alguma operação aplicável ao objeto X;
3. *Campo*, isto é, o objeto Y é um campo do objeto X.

Quando um objeto Y, por exemplo, não é visível apenas ao objeto X, diz-se que o objeto Y tem a *visibilidade compartilhada* com outros objetos. A visibilidade compartilhada, portanto, representa o tipo de compartilhamento onde as estruturas de dados

[3]Processos distintos trabalhando em conjunto e disputando os mesmos recursos computacionais podem provocar estados de inconsistências. Uma maneira encontrada pelos sistemas operacionais para a solução desse problema é criando estados de espera para os processos de maneira que um processo só utilize um recurso após outro já tê-lo utilizado. Essa é a solução encontada para o problema de exclusão mútua.

do objeto Y não são acessadas exclusivamente por um único objeto, ou seja, existe um compartilhamento estrutural.

Enquanto um objeto é uma entidade concreta que existe no tempo e no espaço, uma classe representa somente uma abstração, a *essência* de um objeto. Uma classe é um conjunto de objetos que compartilham da mesma estrutura e das mesmas operações, tendo portanto comportamentos e propriedades comuns. Segundo Peter Wegner em [We89] uma classe é um molde que usa operações do tipo "new" ou "create" definidas nas LPOO, para criar novos objetos.

Existem três tipos básicos de relacionamento entre classes:

Generalização: denota um relacionamento **é-um** entre classes. Ex.: uma *rosa* **é uma** *flor* significando que uma rosa é uma subclasse especializada de uma classe mais genérica - *flor*;

Agregação: denota um relacionamento **parte-de** entre classes. Ex.: uma pétala não é uma flor, mas pétala **é parte de** flor;

Associação: denota alguma conotação semântica entre classes aparentemente não relacionadas. Ex.: *rosas* e *velas* são classes independentes uma da outra, mas que podem representar *coisas* que podem ser usadas para decorar uma mesa de jantar.

As principais estruturas de relacionamentos entre classes utilizadas pelo OOD para suportar os relacionamentos básicos acima são:

1. Relacionamento de Herança: a herança é o tipo de relacionamento entre classes que existe para suportar o relacionamento básico **é-um** de generalização e de associação. Ela estabelece que um relacionamento a esse nível deve compartilhar a estrutura e/ou o comportamento definido em uma ou mais classes. Uma subclasse pode aumentar ou apenas redefinir a estrutura e o comportamento de suas superclasses. A habilidade de uma linguagem suportar esse tipo de relacionamento é que distingue uma linguagem Orientada a Objetos, que o suporta, de uma Baseada em Objetos [We89];

2. Relacionamento de Utilização: esse tipo de relacionamento suporta o relacionamento básico **parte-de** de agregação. Existem dois subtipos diferentes de relacionamentos de utilização:

   (a) *uma interface de classe usa uma outra classe.* Como exemplo desse subtipo apresentamos o relacionamento entre uma classe *livro* e uma classe *biblioteca* que como podemos observar, não *é um* livro, ao contrário livros são **parte-de** biblioteca.

   Utilizando a linguagem Object Pascal [4], desenvolvida pela Apple Computer em

[4] A estrutura e a semântica da linguagem mantêm o mesmo padrão do Pascal. Maiores informações sobre Object Pascal podem ser obtidas em *Macintosh Programmer's Workshop Pascal 3.0 Reference*, 1989, Cupertino, CA: Apple Computer e Schmucker, K., Object-Oriented Languages for the Macintosh, *Byte*, 11(8):179, Agosto de 1986

conjunto com o criador da linguagem Pascal, Niklaus Wirth, podemos escrever a classe biblioteca da seguinte forma:

```
TBiblioteca = object (TObject)
  . . .
  procedure TBiblioteca.inicialize;
  procedure TBiblioteca.verificasaida (UmLivro: TLivro);
  procedure TBiblioteca.verificaentrada (UmLivro: TLivro);
  . . .
end;
```

A classe TBiblioteca como vemos utiliza na definição de sua interface a classe TLivro, dessa maneira a classe TBiblioteca é visível à classe TLivro e a interface ou a implementação dos procedimentos de TBiblioteca podem referenciar a interface de TLivro,

(b) *uma implementação de classe usa outra classe.* Utilizando o exemplo acima, apenas as implementações dos procedimentos da classe TBiblioteca fariam referência à interface da classe TLivro.

Os relacionamentos acima, podem expressar a cardinalidade do relacionamento das classes envolvidas (1:1, 1:n, m:n), de modo semelhante ao que é feito na modelagem de dados através do diagrama E–R [Ch76]. No exemplo acima, biblioteca possui vários livros, entretanto um livro pertence a apenas uma biblioteca. Esse é um exemplo de relacionamento 1:n ;

3. Relacionamento de Instanciação: esse tipo de relacionamento entre classes pode ser também utilizado para suportar os relacionamentos básicos de generalização e associação muito embora de uma maneira diferente do relacionamento de herança. Num relacionamento desse tipo uma instância de uma classe não representa apenas um objeto. Uma instância é uma *coleção de objetos* que formam um conjunto homogêneo se todos os objetos da coleção são da mesma classe, ou heterogêneo se os objetos do conjunto são de classes diferentes mas que compartilham de alguma superclasse comum;

4. Relacionamento Metaclasse: basicamente um relacionamento desse tipo considera uma classe como se fosse um objeto. Assim *metaclasse* é a *classe de uma classe.*

Durante a análise e nos estágios iniciais do OOD, o projetista tem duas tarefas principais:

1. identificar as classes e objetos que formam o vocabulário do domínio do problema;

2. definir as estruturas pelas quais os objetos podem trabalhar em conjunto, para fornecer os comportamentos que satisfaçam os requisitos do problema.

Às classes e objetos que fazem parte do vocabulário do domínio do problema é dado o nome de *abstrações chave* e às estruturas nas quais os objetos trabalham em conjunto para fornecerem algum comportamento que satisfaça a um requisito do problema damos o nome de *mecanismos.*

2.2.4 Classificação

Segundo Grady Booch, a identificação das abstrações chave (classes e objetos) é a parte mais difícil do OOD e ela só é possível devido a dois processos: *descoberta* e *invenção.* Através da *descoberta* são reconhecidas as abstrações chave e os mecanismos que formam o vocabulário do domínio do problema. Através da *invenção* são desenvolvidas outras abstrações mais generalizadas, não criadas numa fase inicial, bem como novos mecanismos para administrar os objetos que podem colaborar entre si.

A classificação é fundamentalmente um problema de encontrar semelhanças. Quando é feita uma classificação, procuram-se grupos de elementos que têm uma estrutura comum ou então que exibem um comportamento comum.

Uma classificação inteligente é mais bem feita, quando ela é construída através de um *processo incremental e iterativo*, isto é, com base em experiências passadas. Assim decide-se criar novas subclasses das classes já existentes (derivação), ou dividir uma classe maior em outras menores (fatoração), ou então criar uma classe maior unindo outras menores (composição). Ocasionalmente, podemos descobrir semelhanças não reconhecidas anteriormente e construir novas classes (abstração).

Segundo Booch, os limites entre *análise* e *projeto* não são bastante claros, muito embora o enfoque seja bastante distinto entre eles. Na análise a intenção é a de descrever *o quê* um sistema deverá ter para obter os resultados pretendidos dentro do domínio do problema. Na Análise Orientada a Objetos (AOO) procuramos modelar o mundo, identificando as classes e objetos que formam o vocabulário do domínio do problema e as responsabilidades do sistema dentro desse domínio. O projeto por sua vez, é orientado na descrição de *como* o sistema desejado irá trabalhar, sem entrar no entanto em detalhes de implementação. Nele são identificadas e definidas as classes e objetos adicionais (não definidos na análise), as abstrações e os mecanismos que fornecem o comportamento que esse modelo requer. Assim é que a AOO deve ser o início mais natural para se utilizar um projeto orientado a objetos (POO).

Como dito anteriormente, a identificação das *abstrações chave* envolve dois processos:

Descoberta: através da *descoberta* são reconhecidas as abstrações usadas no domínio específico do problema;

Invenção: através da *invenção* são criadas novas classes e objetos que não necessariamente fazem parte do domínio do problema, mas que são úteis no projeto ou na implementação do sistema.

Um modo de identificar as *abstrações chave* é olhar para o problema ou projeto em desenvolvimento e verificar se existem abstrações similares às classes e objetos, que já

foram definidos no sistema. Esse é um problema típico de *classificação*. Esta abordagem enfatiza a *reusabilidade* das abstrações e é intrínseca ao processo de criação do OOD.

No OOD primeiro são identificadas as abstrações chave que formam o modelo da realidade. Somente então são adicionados à essas classes os procedimentos que simulam o comportamento da realidade.

2.2.5 Notação do Método OOD

Na Seção 2.2.1 foi apresentado o método OOD como sendo o mais completo dentre os pesquisados em [Mo92]. Para que isso pudesse ser alcançado Grady Booch embutiu no método OOD uma série de mecanismos para representação das múltiplas visões necessárias para capturar todos os detalhes de um sistema complexo, utilizando-se de um rico conjunto de diagramas de representação.

A Figura 2.3 mostra os diferentes modelos e visões capturados pelo método OOD. Eles permitem ao projetista capturar todas as decisões de projeto interessantes e que precisam ser tomadas. Esses modelos são construídos de acordo com as respostas dadas às seguintes questões:

1. Que classes existem e como estas classes estão relacionadas?
2. Que mecanismos são usados para mostrar como os objetos colaboram entre si?
3. Onde cada classe e objeto deve ser declarado?
4. Para qual processador deve um processo ser alocado e para um dado processador, como devem ser escalonados múltiplos processos?

Respostas a essas quatro questões são expressas, respectivamente, através dos seguintes diagramas:

1. Diagramas de Classes
2. Diagramas de Objetos
3. Diagramas de Módulos
4. Diagramas de Processos

Esses quatro diagramas formam a notação básica do OOD e representam a essência da semântica estática do projeto em desenvolvimento. Os dois primeiros diagramas fazem parte da visão lógica do sistema, pois servem para descrever a existência e o significado das abstrações chave que fazem parte do modelo. Os outros dois diagramas fazem parte da estrutura física do sistema porque são usados para descrever módulos e componentes de máquina de uma implementação.

Para expressar a semântica dinâmica do projeto Booch utiliza dois diagramas adicionais que são:

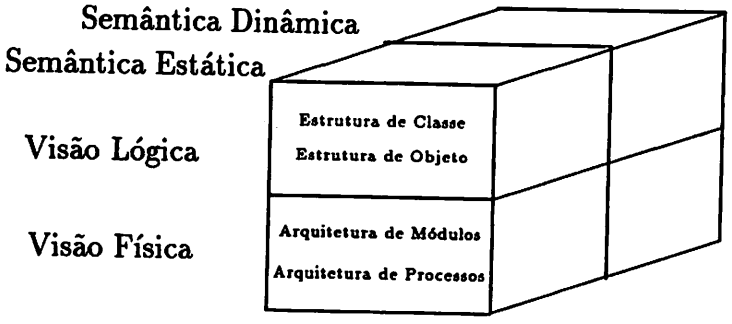


Figura 2.3: Os Modelos do Método de Projeto Orientado a Objetos

1. Diagrama de Transição de Estado (DTE): cada classe deve ter um DTE associado a ela, que indica como a ordenação no tempo de eventos externos pode afetar o estado de cada instância da classe; e

2. Diagrama de Temporização: um Diagrama Objeto representa apenas um instante no tempo de um evento transitório ou configuração de objetos. Isso pode ser imaginado como se uma foto da interação entre os objetos num determinado momento do sistema, fosse tirada. A interação dinâmica dos objetos não pode ser representada, assim o método OOD usa o Diagrama de Temporização em conjunto com cada Diagrama Objeto para mostrar a ordenação do tempo das mensagens, e como elas são enviadas e avaliadas.

Diagramas de Classes

Um Diagrama de Classe é usado para mostrar a existência de classes e seus relacionamentos no projeto lógico de um sistema. Os três mais importantes elementos de um Diagrama de Classe são:

Classes: as linhas descontínuas que delimitam as fronteiras da classe indicam que os usuários desta classe geralmente operam sobre as instâncias da classe, não sobre a classe em si (Figura 2.4).

O nome da classe é obrigatório e deve ser colocado dentro do ícone. Além disso todo nome de classe deve ser único.

Relacionamentos entre Classes: como exposto anteriormente os relacionamentos entre classes podem ser de herança, utilização, instanciação e metaclasse. Os ícones expressando esses relacionamentos são mostrados na Figura 2.5.

O relacionamento de Utilização é representado por uma linha dupla com um círculo colocado junto à classe que usa os recursos de outra classe (um círculo cheio diz que a implementação da classe utiliza os recursos da outra classe; um círculo vazio diz que a interface da classe utiliza os recursos da outra classe).

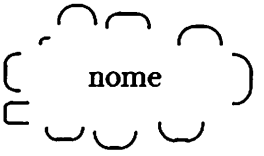


Figura 2.4: Ícone para Classe

O relacionamento de Herança e de Instanciação são representados por uma linha simples, exceto que para o relacionamento de herança a linha usada é sólida.

O relacionamento Metaclasse é representado por uma linha dupla descontínua e para um relacionamento dito *indefinido* utilizamos uma linha simples tracejada sem seta. Esse último, é o tipo de relacionamento provisório que ocorre nos estágios iniciais do desenvolvimento quando ainda não temos certeza dos tipos de relacionamentos que irão ser utilizados.

Quando instâncias de uma subclasse não são de tipos compatíveis com as instâncias da superclasse, isto é, a subclasse redefine as operações e/ou estruturas da superclasse, então colocamos uma linha perpendicular à linha do relacionamento, do lado oposto à seta do relacionamento.

A representação do grau de relacionamento entre as classes (cardinalidades) também pode ser mostrada, colocando-se o ícone que representa a cardinalidade da classe no relacionamento próximo à sua classe (Figura 2.5).

Classe de Utilitários: uma classe de utilitários representa um subprograma livre ou um conjunto de subprogramas livres.

O ícone para representar esse tipo de classe é idêntico ao da classe, exceto por um sombreado na parte inferior do ícone (Figura 2.6).

O nome da classe de utilitários é obrigatório e deve ser único, devendo ficar dentro do ícone.

O Apêndice B – Esquemas – descreve uma maneira diferente de representação do modelo desenvolvido pelo OOD, além da notação diagramática, para representar as várias estruturas do modelo de objetos. Segundo Booch é preciso algo mais além de gráficos para fornecer meios para a documentação completa do significado de cada elemento do modelo de objetos.

Diagramas de Transição de Estados

Um diagrama de classe não mostra como as instâncias dessas classes individuais comportam-se dinâmicamente. O comportamento dinâmico dos objetos de classe é documentado através do uso dos Diagramas de Transição de Estados - DTE.

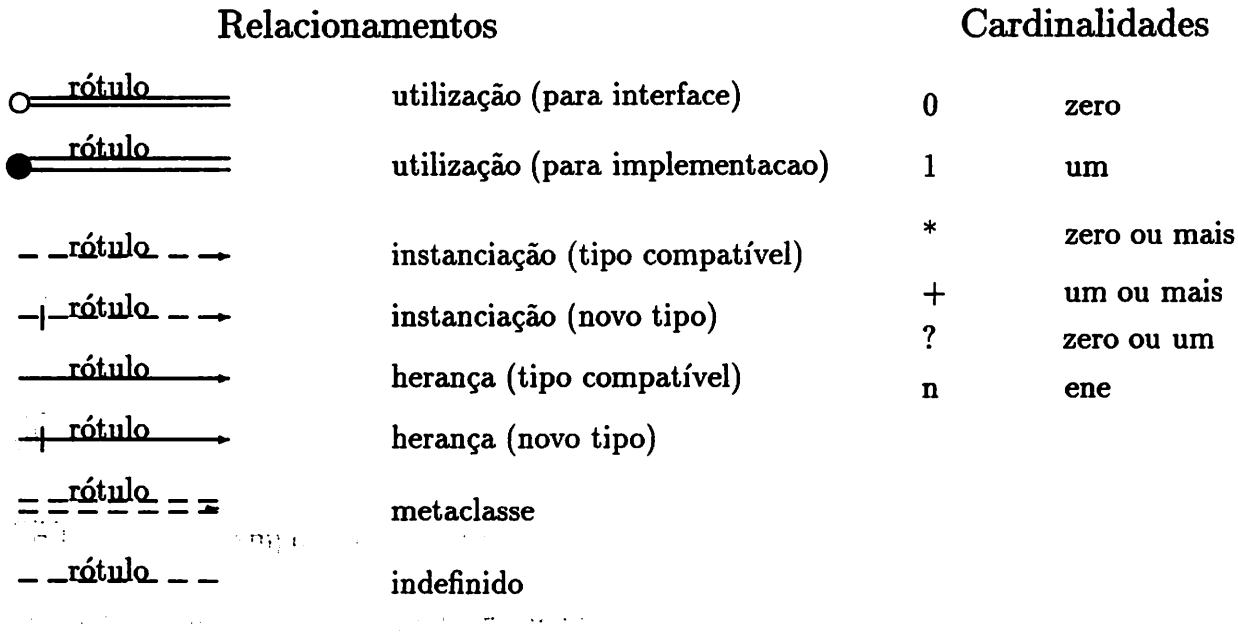


Figura 2.5: Ícones para Relacionamentos de Classe e Cardinalidades

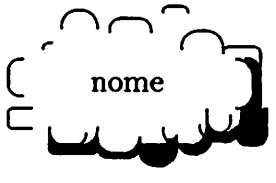


Figura 2.6: Ícone para Classe de Utilitários

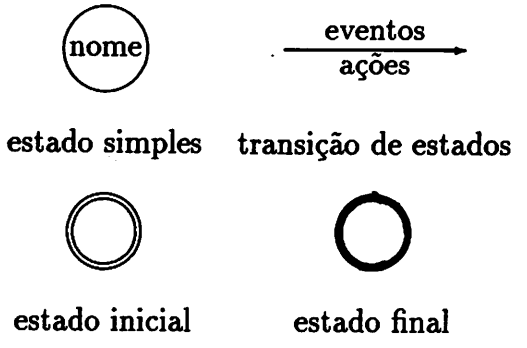


Figura 2.7: Ícones para Diagramas de Transição de Estados

Esses diagramas mostram o estado de uma classe, os eventos que causaram a transição de um estado para outro, e as ações que resultam da mudança de estado.

Um Diagrama de Transição de Estados está intimamente relacionado com as outras partes da notação: o Esquema de uma classe pode incluir o Diagrama de Transição de Estados, e as ações descritas em um Diagrama de Transição de Estados particular podem apontar para outros Diagramas de Objetos.

Os principais conceitos e símbolos relacionados com um Diagrama de Transição de Estados são:

Estados: Um círculo representa um estado simples. Dentro do ícone de estado deve ser colocado o nome do estado que deve ser único para um dado Diagrama de Transição de Estado (Figura 2.7).

Um *estado inicial* é representado por um círculo de linha dupla vazio, e o estado final por um círculo duplo cheio (Figura 2.7).

Transições de Estado: Um relacionamento que é significativo entre os estados no diagrama é chamado de *transição de estados.* Uma transição de estados pode ocorrer entre dois estados diferentes ou num mesmo estado.

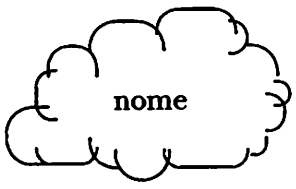
Uma transição de estado é representada por uma linha direta entre dois estados e deve ser rotulada com pelo menos um evento que ocasionou a transição (Figura 2.7).

Uma definição para *evento* e para *ação* no método OOD pode ser encontrada em [Ch92]. Nele *evento* é descrito como sendo:

- uma mudança no estado do objeto, alguma coisa que acontece em um instante determinado no tempo. Assim um evento pode ser produzido por sucessivas transições na máquina de estados de um objeto.

- um evento *externo* que age sobre um objeto qualquer, estabelecendo uma condição para que haja uma transição de estados.

A esses dois tipos de evento, um evento *interno* devido a mudança de estado e um evento *externo* que é percebido por um objeto, Booch em [Bo91] dá os nomes, respectivamente, de **ação** e de **evento.**



lista de mensagens (dentro do sistema)

rótulo (fora do sistema)

Figura 2.8: Ícones para Objetos e Relacionamentos de Objetos

Eventos e Ações não precisam ser rotulados, dentro de um mesmo diagrama, com nomes únicos pois um mesmo evento pode ocasionar uma transição para muitos estados diferentes, e uma mesma ação pode ser resultado de muitas transições diferentes.

Diagramas de Objetos

Os Diagramas de Objeto são usados para mostrar a existência de objetos e seus relacionamentos no projeto lógico de um sistema. Um simples diagrama de objeto representa todo ou parte da Estrutura de Objeto de um sistema.

As classes são mais estáticas no projeto de um sistema, comparado aos objetos que são mais transitórios e que muitas vezes podem ser criados ou destruídos durante a execução de um programa.

Todo objeto em um diagrama de objeto denota alguma instância de uma classe. As operações usadas em um diagrama de objeto devem ser consistentes com as operações definidas nas classes associadas.

Os Diagramas de Classe documentam as *abstrações chave* no sistema, ao passo que os Diagramas de Objeto documentam os *mecanismos* que manipulam essas abstrações. Os dois elementos mais importantes de um Diagrama de Objeto são:

Objetos: a Figura 2.8 mostra o ícone que usamos para representar um objeto. Este ícone é similar ao de uma classe exceto que a linha que delimita o ícone é sólida, mostrando que ela representa uma "instância de", e não uma classe. O nome do objeto é obrigatório mas não precisa ser único, pois em um sistema pode haver várias instâncias de um mesmo objeto. As propriedades de um objeto devem aparecer no canto inferior esquerdo da mesma maneira que para as classes. As propriedades mais importantes de um objeto são a *concorrência* e a *persistência* de cada objeto.

Relacionamentos de Objeto: um relacionamento entre dois objetos simplesmente significa que os objetos podem enviar mensagens uns para os outros. Desde que as mensagens são bidirecionais, usamos linhas diretas sem seta para designar estes relacionamentos (Figura 2.8). Uma linha sólida representa relacionamentos de objetos que podem ser modelados *dentro* do sistema e uma linha pontilhada para aqueles relacionamentos que são *exteriores* ao sistema.

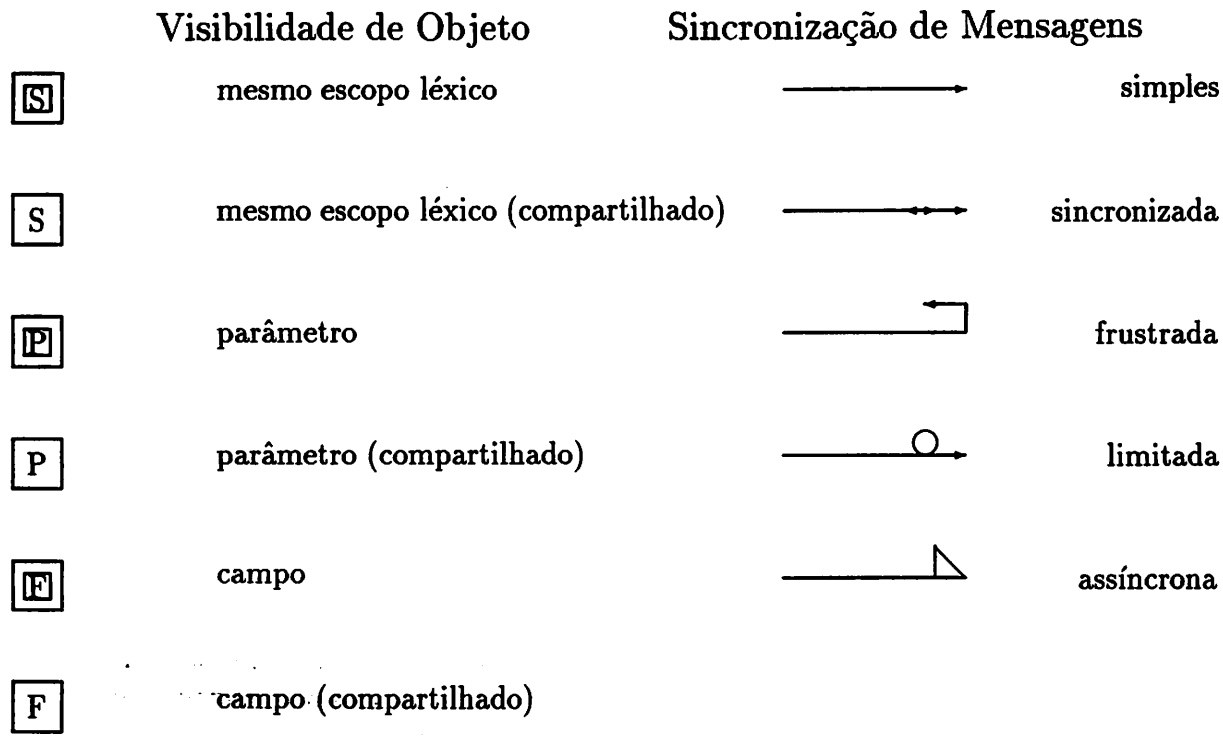


Figura 2.9: Ícones para Visibilidade de Objetos e Sincronização de Mensagens

Continuando o estudo da notação gráfica do Diagrama de Objeto, podemos adicionar maiores detalhes à sua representação de duas maneiras:

Visibilidade do Objeto: documentamos como dois objetos veem um ao outro através dos ícones apresentados na Figura 2.9. Como observamos existem seis maneiras pelas quais um objeto pode ser visível a outro objeto. Por exemplo se um objeto R é um campo compartilhado de um objeto S, então ilustramos essa espécie de visibilidade colocando o ícone para campos compartilhados na linha que representa o relacionamento entre os objetos (R e S), próximo ao objeto R.

A fim de melhorar o entendimento sobre a visibilidade dos objetos, podemos arranjá-los dentro dos Diagramas de Objeto de maneira a colocar os objetos ativos (atores) no topo do diagrama e os objetos passivos (servidores) na base do diagrama de objeto. Similarmente podemos representar o relacionamento de composição colocando um ícone objeto dentro de outro.

Sincronização de mensagem: igualmente importante seria mostrar como objetos relacionados interagem entre si. Para objetos puramente seqüenciais uma simples linha direta desenhada sobre a linha de relacionamento é o suficiente. Entretanto a coisa se complica quando temos múltiplas linhas de controle. No total existem cinco diferentes espécies de sincronização de mensagens conforme exposto abaixo:

1. Simples: usada para objetos puramente seqüenciais ou seja, apenas uma linha de controle;

2. Sincronizada: uma operação começa somente quando o emissor tiver iniciado a ação e o receptor esteja pronto para aceitar a mensagem; o emissor e o receptor irão esperar indefinidamente até que ambos estejam prontos para prosseguir;
3. Frustrada: similar à sincronizada, exceto que o emissor irá abandonar a operação se o receptor não estiver pronto imediatamente;
4. Limitada: similar à sincronizada, exceto que o emissor irá esperar por uma quantidade de tempo determinado para que o receptor fique pronto;
5. Assíncrona: um emissor deve iniciar uma ação independentemente de o receptor estar esperando a mensagem ou não.

Os ícones para a visibilidade de objetos e sincronização de mensagens estão representados na Figura 2.9, sendo que os ícones para sincronização de mensagens devem ser rotulados com a lista de mensagens.

Diagramas de Temporização

Os Diagramas de Objetos são estáticos, isto é, mostram apenas uma coleção de objetos que passam mensagens uns para os outros, mas não mostram o fluxo de controle, nem a ordenação dos eventos no tempo. Isso também acontece com o Diagrama de Transição de Estados que mostra apenas uma representação das mudanças de estados dentro dos objetos, mas não entre um conjunto de objetos que se cooperam.

Para auxiliar na visualização da ordenação dos eventos dentro do OOD, Booch utiliza o Diagrama de Temporização. Na Figura 2.10 observamos esse diagrama, onde no eixo das abcissas (X) é representado a variável *Tempo* e no eixo das ordenadas (Y) os objetos que interagem. O Tempo pode ser expresso em números absolutos ou relativos e cresce da esquerda para a direita.

No eixo dos Y's devemos colocar apenas os objetos cuja interação é de interesse para o desenvolvimento do sistema.

As linhas tracejadas no Diagrama de Temporização indicam um aninhamento dinâmico das mensagens, isto é, se uma operação qualquer se encerra o controle deve retornar à operação do objeto que possui a linha tracejada unida a ela. Os símbolos "*" e "!" mostram, respectivamente, a criação e a destruição de um objeto.

Diagramas de Módulos

Diagramas de Classes e Diagramas de Objetos são usados para documentar *projetos lógicos* de um sistema. Para a implementação efetiva do projeto lógico, devemos utilizar a noção de *projeto físico.*

O Diagrama de Módulo é usado para mostrar a alocação de classes e objetos em módulos, no projeto físico de um sistema. Um módulo pode representar toda ou apenas parte da arquitetura de módulo de um sistema. Os principais conceitos para esse tipo de diagrama são:

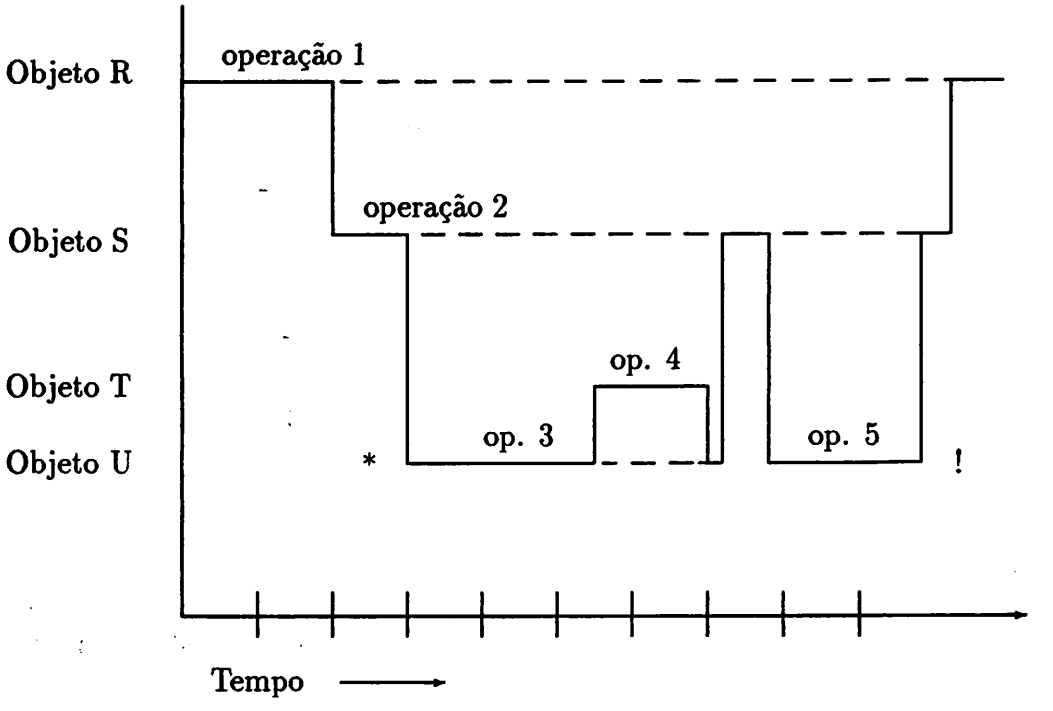


Figura 2.10: Ícone para Diagrama de Temporização

Módulos: a Figura 2.11 representa as diferentes espécies de módulos em um Diagrama de Módulo. O nome do módulo é obrigatório e deve ser colocado no topo do ícone. Todo nome de módulo deve ser único dentro de um subsistema (subsistema para módulos é o mesmo que categoria de classes para classes, isto é, módulos relacionados agrupados em módulos de módulos).

Visibilidade de Módulos: o único relacionamento que existe entre dois módulos é o de *dependência de compilação*, representado por uma linha direta entre os módulos que possuem essa dependência, com a seta apontando para o módulo da qual o outro módulo depende (Figura 2.11).

Diagramas de Processos

Em sistemas de grande porte deve-se às vezes projetar múltiplos subsistemas, que por vezes são executados em mais de um computador; isto requer decisões de projeto diferentes daquelas utilizadas para identificar classes e objetos. Considerando esse fato, Booch introduziu o conceito de Diagramas de Processos, que têm por objetivo a visualização do problema de alocação de processos nos processadores dentro do projeto físico de um sistema.

Um processo pode representar toda ou apenas parte da arquitetura de processo de um sistema.

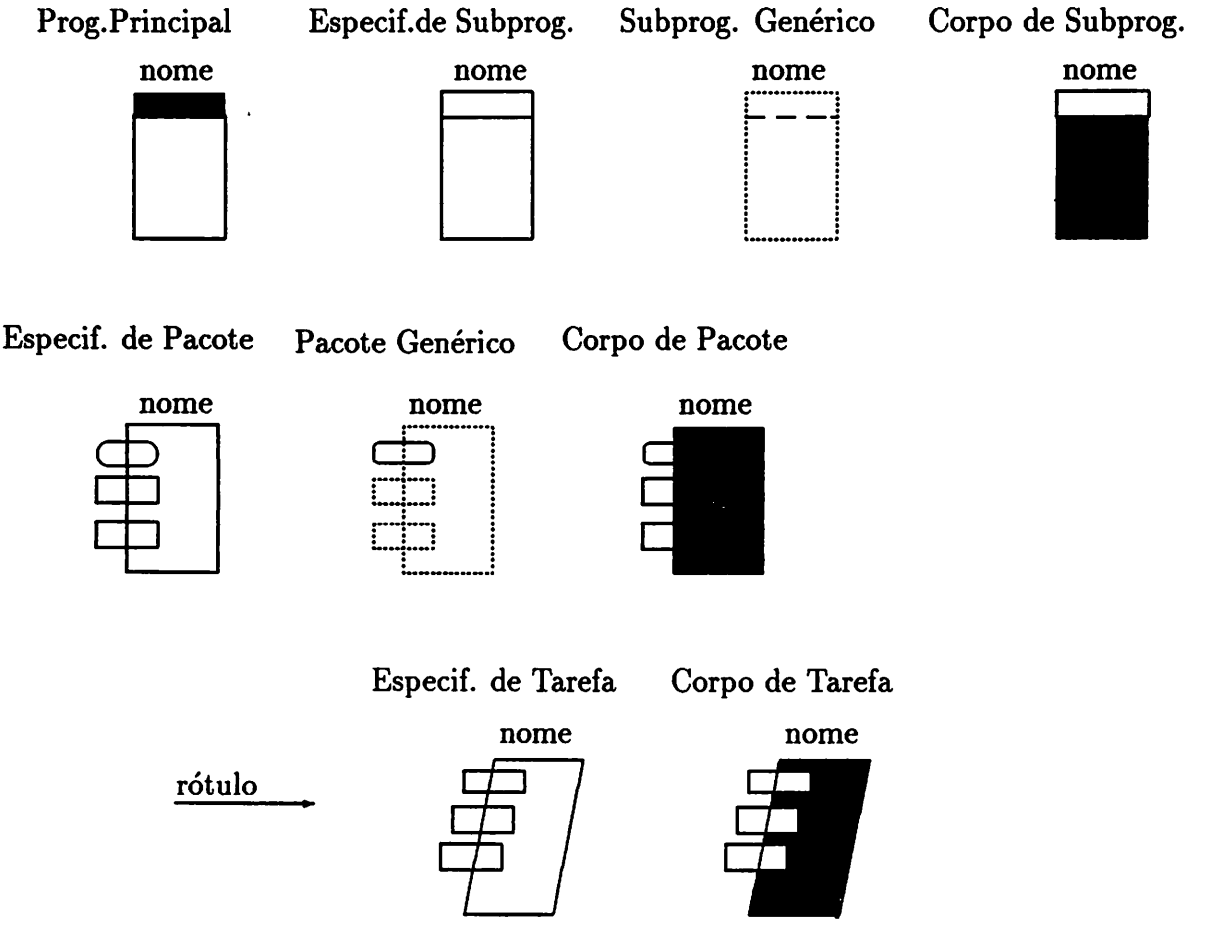


Figura 2.11: Ícones para Módulos e Visibilidade de Módulo

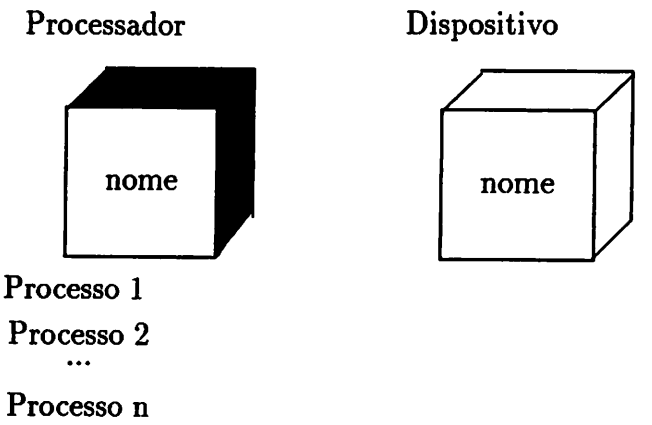


Figura 2.12: Ícones para Processadores e Dispositivos

Mesmo em sistemas que são executados em um processador único o Diagrama de Processo é útil quando a implementação envolve mais de um dispositivo ou objeto ativo, significando que podem existir múltiplos processos de uma só vez.

Os três mais importantes elementos de uma arquitetura de processo no modelo objeto são os processadores, os dispositivos, e as conexões.

Um *processador* é uma peça do componente físico capaz de executar programas, ao contrário do *dispositivo* que não é capaz de executar programas. Um *dispositivo* pode ser, por exemplo, uma impressora, um monitor de vídeo, etc. O nome do processador e do dispositivo são obrigatórios.

Uma *conexão* representa um acoplamento direto entre um dispositivo qualquer e um processador ou entre um processador e outro processador ou entre um dispositivo e outro dispositivo, e é representada por uma linha direta entre os elementos do diagrama de processo (Figura 2.12).

Para fazer o escalonamento de execução dos processos dentro de um processador, Booch enumera cinco maneiras:

Preemptivo: processos de mais alta prioridade que estejam prontos para serem executados, devem ocupar o processador em lugar de processos de mais baixa prioridade correntemente sendo executados. Processos de igual prioridade ganham uma fatia igual de tempo do processador, de maneira a distribuir uniformemente os recursos computacionais;

Não Preemptivo: o processo corrente continua sendo executado até que ele abandone o controle do processador;

Cíclico: o controle passa de um processo para outro, sendo que a cada processo é dado uma quantidade fixa de tempo de processamento, processo esse conhecido como *janela de tempo*;

Executivo: algum algoritmo controla o processo de escalonamento;

Manual: os processos são escalonados por um usuário externo ao sistema.

Similarmente ao método JSD (página 12), o método OOD também apresenta etapas do desenvolvimento ligadas às fases de implementação e estruturação física do sistema, representadas no modelo de objetos (Figura 2.3) pelas arquiteturas de módulos e de processos. Essas fases são agrupadas no modelo de objetos com o nome de *Visão Física* do projeto.

Capítulo 3

Uma Comparação entre os Métodos JSD e OOD

Nesse capítulo faremos um paralelo entre os dois métodos, comparando-os em relação a alguns conceitos importantes do modelo de objetos.

Na Seção 3.1 fazemos uma comparação dos conceitos mais característicos do modelo objeto, presentes no OOD, e que de alguma forma também se apresentam no método JSD. Aproveitamos essa seção, e apresentamos algumas considerações sobre o tipo de representações gráficas utilizadas pelos dois métodos.

Na Seção 3.2 apresentamos outros conceitos de grande importância do modelo de objetos, presentes no OOD e que não são definidos no método JSD.

3.1 Conceitos importantes presentes nos dois métodos

Classes e Objetos: no modelo desenvolvido que surge do OOD, um *objeto* é considerado como sendo uma *instância de uma classe*, identificado de modo único, possuindo um *estado* (valores das variáveis locais) e um *comportamento*. Uma *classe* por sua vez, é definida como sendo uma abstração de um ou mais objetos, onde são definidas as *operações* (métodos) e as *estruturas* (variáveis locais) que são manipuladas exclusivamente por essas operações.

O método JSD define o conceito de *entidade* que pode ser comparado ao conceito de *objeto* do OOD, isto é, modela-se uma entidade como sendo uma abstração da realidade sobre a qual mensagens (eventos) são enviados, a fim de se obter em resposta uma *ação* (comportamento) e que possui variáveis locais determinantes de seu estado (*vetor de estados*). Comparado ao conceito de classe, o método JSD define o conceito de *tipo-entidade* que é o local onde as ações, a ordenação no tempo das ações, e as variáveis locais são declaradas.

Em termos de modelagem, as mesmas regras para a descoberta de objetos do modelo orientado a objetos são válidas para a descoberta das entidades do modelo JSD.

Abstração: outro conceito importante no modelo de objetos é o de *abstração*, sua utilização é considerada fundamental para a construção dos modelos orientados a objetos. Em [In91], Darrel Ince chega a afirmar que "A abstração de tipos de dados (···) é parte tão fundamental das linguagens de programação orientadas a objetos, que muitos consideram o desenvolvimento de sistemas de software usando tais linguagens, como sendo simplesmente uma manipulação de tais abstrações (···)"

O método JSD também utiliza-se da abstração para a definição dos elementos que irão compor o modelo desenvolvido por ele, sendo ela a responsável pela descoberta de todas as entidades que irão compor a estrutura principal do modelo.

Encapsulamento: para que um modelo seja considerado orientado a objetos, Booch em [Bo91] coloca o encapsulamento como um de seus fatores fundamentais. O encapsulamento é a capacidade de ocultar as estruturas que os elementos que formam a base do método (objetos no OOD) possuem. Desse modo as variáveis locais que formam a estrutura dos elementos não são livremente acessadas pelos outros elementos do modelo. O seu acesso se dá apenas por meio do envio de mensagens de um elemento a outro. Em resposta, o segundo elemento age sobre as suas estruturas através das operações definidas em suas classes e disponíveis para seu uso, alterando o seu estado.

Tanto em OOD quanto em JSD o encapsulamento se apresenta de uma forma intrínseca, mas dependem da forma como é estabelecida a comunicação entre os elementos ou declaradas as variáveis, permitindo ou não o livre acesso às suas estruturas. Assim no método JSD a *conexão por inspeção do vetor de estados* permite que uma entidade (a que inspeciona) obtenha os valores das variáveis locais de outra entidade (a que é inspecionada), sem que seja possível alterar os seus valores. No OOD as variáveis declaradas como *públicas* podem ser acessadas por meio das operações disponíveis em objetos de outras classes.

Identificação: a identificação das instâncias das classes no OOD e dos tipos-entidade em JSD é feita de modo a identificar de maneira única os elementos que são a base dos dois métodos, isto é, os *objetos* e as *entidades* respectivamente. Nos dois métodos, a preocupação com o processo de identificação tem lugar nas partes finais do desenvolvimento do sistema, no chamado *projeto físico* do modelo.

A identificação dos objetos no OOD existe a partir da instanciação das classes às quais os objetos pertencem, e que têm a responsabilidade na criação desses objetos. As linguagens orientadas a objetos (LOO) normalmente determinam duas etapas para a criação e identificação dos objetos:

1. declaração do nome das variáveis que irão identificar os objetos de uma determinada classe. Ex.: item1, item2: TPalleteItem ;
2. criação dos objetos em si, isto é, separação de um espaço de memória identificada de modo único pelo nome da variável do item anterior. Ex.: para a LOO C^{++} new(item1), new(item2);

O processo de identificação no método JSD, isto é, o processo de criação das entidades, é realizado na fase VI, Implementação, utilizando o conceito de Separação do Vetor de Estados, que é a separação do texto dos processos dos respectivos vetores de estados. Dessa forma um processo que é instanciado mais de uma vez, necessita apenas de uma cópia do texto do processo; o vetor de estado das instâncias são passados como parâmetros pelo processo escalonador quando da ativação do processo.

Em JSD as entidades são identificadas pelos *atributos de entidade* ([Su88]), sendo que os identificadores devem possuir um nome significativo e devem ser um campo chave da estrutura que armazena o vetor de estados, não podendo ser atualizado pelas ações da própria entidade.

O conjunto das medidas, separação do vetor de estados e identificação das entidades por um campo chave desse vetor, é parte intrínseca do processo de identificação única do modelo JSD.

Visibilidade: em [Bo91], Booch afirma: "Por visibilidade, entendemos a habilidade de uma abstração *enxergar* outra e assim referenciar recursos de outras visões externas". Em JSD bem como no OOD a visibilidade é questão principalmente do modo como os relacionamentos entre os elementos se processam.

No OOD vimos que existem seis maneiras de um objeto *enxergar* outro objeto (Figura 2.9).

Em JSD podemos considerar a existência de dois tipos de visibilidade, determinadas principalmente pelos dois tipos de conexões existentes no método. O relacionamento criado pela *conexão por mensagens*, que permite uma visão dos dados de saída da *entidade emissora*, pela *entidade receptora* através de um buffer de memória (canal de mensagens) e o relacionamento criado pela *conexão por inspeção do vetor de estados* que permite uma visibilidade total nos dados por parte da *entidade que inspeciona*, sem que ela entretanto possa alterar os valores do vetor de estado da *entidade inspecionada*. A esse tipo de conexão está relacionada a operação definida como *seletora* no OOD, realizada por um objeto sobre outro (página 16).

A visibilidade dos dados obtida pela conexão por inspeção do vetor de estados, pode ser considerada menos restritiva se comparada à visibilidade obtida pelo OOD, devido ao fato de os dados de uma entidade, na conexão JSD, serem inspecionados diretamente por outra entidade; ao passo que no OOD um objeto só tem acesso às estruturas de outros objetos através das operações definidas nesses objetos, ou seja, os dados no OOD podem ser considerados mais encapsulados do que no método JSD.

Relacionamentos: um objeto dentro de um modelo orientado a objetos não vive isoladamente, isto é, um sistema para poder representar bem um modelo da realidade deve ser formado por objetos que interagem entre si, de modo a poderem simular o domínio do problema no qual estamos interessados. Essa interação pode ser definida

como o ato de unir duas ou mais *coisas* ou *objetos*, com o intuito de aumentar o poder de expressão dos elementos que se relacionam.

No OOD, quando dois objetos, ou duas classes, ou um objeto e uma classe, possuem alguma ligação entre si, dizemos que existe um relacionamento entre eles. Em JSD quando duas entidades trocam informações por meio dos dois tipos de conexão existentes - mensagens ou inspeção do vetor de estados - dizemos que existe um relacionamento entre elas.

Em [Su88], Sutcliffe afirma que quando duas ou mais entidades, no método JSD, compartilham de uma ou mais *ações comuns*, elas criam uma ligação lógica entre as entidades que a compartilham. Em conseqüência, essa propriedade, além de estabelecer os relacionamentos entre as entidades, também expressa uma restrição de integridade. Em um sistema de controle de empréstimos de uma biblioteca, por exemplo, uma ação *emprestar* é comum às entidades *livro* e *leitor* e deve estabelecer além da existência de um relacionamento entre elas, também uma restrição de integridade de leitor para livro já que a entidade *livro* não pode executar ou sofrer uma ação *emprestar* antes que a entidade *leitor* tenha executado a ação de tomar emprestado um livro.

Semelhante ao OOD, o método JSD desenvolve o modelo do sistema fazendo uso de conexões entre processos (por mensagem e/ou por inspeção do vetor de estados), numa forma bem aproximada aos relacionamentos de utilização usados no desenvolvimento dos modelos objetos se compararmos o que afirma Grady Booch em [Bo91] à página 88: "(···) um relacionamento de utilização (···) significa que mensagens devem passar por esse caminho[1]". Dessa maneira, as informações que são passadas de uma entidade a outra por meio do envio de mensagens, representadas no modelo JSD pela *conexão por mensagens*, nada mais são do que a expressão do relacionamento de utilização do OOD.

Uma semelhança observada no processo de relacionamentos dos métodos JSD e MPOO, é a sentida quando da conexão por envio de mensagens do JSD e a da passagem de mensagens entre objetos seqüenciais do OOD, que forçam uma aproximação entre os elementos conectados (interdependência entre elementos), e que provocam uma sincronização entre os processos dos dois métodos, isto é, toda mensagem enviada pelo processo emissor deve obrigatóriamente, ser lida, na ordem de chegada, pelo processo receptor.

Outro relacionamento que existe entre os objetos do OOD é o *relacionamento de composição* descrito à página 16. O método JSD representa entidades que agrupam outras entidades (*entidades marsupiais*) através da comunicação entre processos, como podemos observar no exemplo dado por M. Jackson em [Ja83], e que de uma maneira aproximada servem para representar os relacionamentos de composição do OOD. O exemplo cita uma entidade *cliente*, possuindo várias contas em um banco. Como fazer para manipular as várias contas do cliente? Após a análise do problema

[1] Por caminho entende-se o canal de comunicação existente entre dois objetos.

surge a necessidade de separação da entidade *cliente* em duas ou mais espécies de entidades. Uma entidade *cliente* e uma ou mais entidades *conta*, uma para cada conta bancária do cliente; dessa forma um cliente pode possuir mais de uma conta em um mesmo banco. Esse processo do JSD como afirmamos pode ser utilizado para a representação do relacionamento de composição.

O relacionamento de composição como feito acima no método JSD, através das entidades marsupiais, admite apenas a composição por instâncias de um mesmo tipo-entidade, ou seja, por meio de *entidades homogêneas.* Ao contrário, o relacionamento de composição do OOD permite também que se formem objetos a partir da composição de outros objetos de classes diferentes, isto é, por meio de *objetos heterogêneos.* Comparando os dois tipos de relacionamento por composição podemos observar que o método JSD restringe mais o poder de modelagem das linguagens **OO** em relação aos métodos que desenvolvem o modelo de objetos.

Diagrama de Transição de Estados: o DTE no modelo do OOD pode representar:

1. os vários estados em que um objeto pode se encontrar, enquanto aguarda um evento externo (mensagem) que produza como resposta uma mudança de estado;
2. o(s) evento(s) que gerou(raram) uma mudança de estado;
3. a ação ou ações (evento(s) interno(s) segundo [Ch92]) geradas sobre outros objetos quando da mudança de estado.

No JSD o DEC representa:

1. a ordenação no tempo das ações que constituem a entidade,
2. as ações ou componentes elementares, segundo [Su88], representados como folhas de uma árvore hierárquica,
3. os atributos ou operadores elementares, segundo [Su88], que são ou os identificadores (campos chaves) ou então as propriedades que representam a vida de uma entidade.

Diferenças encontradas nos diagramas que representam a visão dinâmica do modelo lógico dos dois métodos:

- Basicamente o que é representado no DEC do método JSD, também pode ser representado no DTE, apenas observando que no DEC tanto as ações (ou métodos conforme o modelo de objetos) requeridas quanto as sofridas são explicitamente representadas no diagrama, ao passo que no DTE apenas os métodos implementados pelo próprio objeto são representados;
- Outra diferença importante encontrada na representação dos dois diagramas é que no DTE a ordem de execução das operações não pode ser mostrada,

existindo apenas um ícone para representar o estado inicial e outro para representar o estado final. Para suprir a falta desse sequenciamento das operações dos objetos, o modelo do OOD se utiliza do Diagrama de Temporização.

3.2 Conceitos importantes ausentes no método JSD

Herança: é um dos principais conceitos do modelo de objetos, e principal responsável pela reutilização dos objetos no desenvolvimento de sistemas através da hieraquização estrutural e funcional dos elementos de um modelo. Utilizada principalmente para a modelagem do domínio do problema através da decomposição orientada a objetos. Dentre as várias definições encontradas nos livros para o conceito de herança a exibida em [Co91] pode ser considerada uma das mais simples. Segundo ele herança é "O mecanismo para expressar similaridades entre classes, simplificando a definição de classes similares àquelas previamente definidas". Booch apresenta a herança, um dos quatro tipos de relacionamentos entre classes do OOD (página 18), como sendo o conceito mais importante para que o sistema gerado por um modelo possa ser considerado orientado a objetos.

O método JSD não trata as entidades do modelo como estruturas hierárquicas impossibilitando, dessa maneira, o seu tratamento por meio do mecanismo de herança. Esse é considerado como o principal motivo para que pesquisadores da área afirmem ser o JSD apenas um método *baseado em objetos.*

Polimorfismo: a palavra polimorfismo se origina do grego *polis* (muitos) e *morphos* (formas), isto é, ser polimórfico significa então, ter várias formas. Assim como na identificação esse item também entra no processo de modelagem do sistema apenas em uma etapa adiantada do desenvolvimento - a do projeto físico, onde as preocupações com o ambiente de desenvolvimento (software e hardware) são levadas em consideração. Segundo Booch "O polimorfismo representa um conceito em teoria dos tipos, na qual um simples nome (uma declaração de variáveis por exemplo) pode denotar objetos de muitas classes diferentes que estão relacionadas por alguma superclasse comum". O polimorfismo existe devido às características de *herança* e de *ligação dinâmica* que interagem[2]. A idéia básica é a de que se Y herda de X, Y "é um" X, e portanto em qualquer lugar que uma instância de X é esperada, uma instância de Y também é permitida.

Essa característica está muito mais ligada à capacidade das linguagens de programação em fornecer este tipo de tratamento ao modelo a ser implementado, do que propriamente um recurso a ser utilizado nas fases anteriores à codificação (análise e projeto) no ciclo de vida de construção de um sistema.

O método JSD não trata desse tipo de problema devido não ser da época de seu desenvolvimento o oferecimento desse tipo de recurso pelas linguagens de programação

[2]Por ligação dinâmica entendemos o mecanismo das LOO que possibilita a todos os tipos de todas as variáveis e expressões serem conhecidos apenas em tempo de execução.

(COBOL, FORTRAN, PASCAL, etc), além do que, como visto, o JSD apesar de se preocupar em adaptar o modelo aos recursos computacionais disponíveis (Fase VI), não se preocupa com a codificação do modelo em si.

Capítulo 4

Método JSD-OO

Como observado, o JSD apesar de ser um método de aprendizado rápido, de poucos conceitos simples e de fácil entendimento, ainda não se tornou **popular** no sentido de larga utilização e aceitação pela comunidade. Com o crescente interesse no paradigma de orientação a objetos, ele tem recebido mais atenção, principalmente por parte da comunidade acadêmica [Bo86], pois fica patente a semelhança dos conceitos de JSD com os conceitos utilizados na modelagem por objetos. Entre os conceitos que podem ser considerados semelhantes, podem ser destacados: a modelagem inicial do sistema feita a partir do domínio do problema com base nos conceitos de entidade e ação, a estruturação dos elementos que constituem o domínio de modo a agrupar as variáveis locais próximos às suas ações (encapsulamento), etc.

O Método JSD-OO (Jackson System Development - Object Oriented) [Al93] deve ser considerado como uma extensão do método JSD naquilo que lhe falta para se tornar um método de projeto orientado a objetos, dentro desse paradigma. A premissa é a de que essa extensão possa vir a ajudar a divulgar ainda mais o método JSD, e a torná-lo mais popular pois ele possui os requisitos básicos para isso: simplicidade e economia de conceitos, e notação simples e concisa para representação dos mesmos.

Assim como em outros métodos orientados a objetos [Co91], o JSD-OO deve especificar os seus modelos, nascidos da aplicação do método, de modo poderem ser codificados em uma linguagem de programação que utilize os mecanismos baseados no paradigma de orientação a objetos. Só assim o código gerado pela linguagem pode tirar o máximo de proveito do projeto gerado da aplicação do JSD-OO.

4.1 Notação do Método JSD-OO

Para manter o máximo de compatibilidade entre o JSD e o JSD-OO, a maioria dos conceitos do projeto lógico JSD deverão ser aproveitados, sendo alguns adaptados e outros incluídos onde isso se fizer necessário, alcançando assim o nível de adequação desejada. As principais alterações e recursos considerados são:

- *Classes e Objetos.* No modelo que surge do OOD, um **objeto** é considerado como

sendo uma **instância de uma classe**, identificado de modo único, e possuindo um **estado** (valores das variáveis locais) e um **comportamento**. Uma **classe** por sua vez, é definida como sendo uma abstração de um ou mais objetos, onde são definidas as **operações** (métodos) e as **estruturas** (variáveis locais) comuns, e que são manipuladas exclusivamente por essas operações.

No JSD, como já dito, o conceito de **objeto** do OOD pode ser comparado ao conceito de **entidade**, isto é, modela-se uma entidade como sendo uma abstração da realidade sobre a qual mensagens (eventos) são enviadas, a fim de se obter em resposta uma **ação** (comportamento) e que possui variáveis locais determinantes de seu estado (**vetor de estados**). Semelhante ao conceito de **classe** no OOD, o método JSD define o conceito de **tipo-entidade** onde as ações e a sua ordenação no tempo são especificadas, e as variáveis locais são declaradas.

No JSD-OO o símbolo para tipo-entidade será representado como mostra a Figura 4.1.

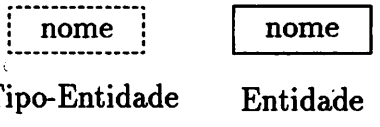


Figura 4.1: Ícones para Tipo-Entidade e Entidade no JSD-OO

A representação de tipo-entidade por um retângulo de arestas descontínuas tem o mesmo objetivo do método OOD ao representar uma classe de objetos com uma *nuvem* de arestas também descontínuas, indicando que os clientes do tipo-entidade atuam sobre suas instâncias, isto é, sobre as entidades do modelo e não sobre o tipo-entidade em si. Em termos de modelagem, as mesmas regras para a descoberta de objetos do modelo orientado a objetos são válidas para a descoberta das entidades do modelo JSD.

- *Encapsulamento.* Esse conceito, também denominado por alguns autores como *ocultamento de informação*, é definido em orientação a objetos através da regra de que a interface para cada objeto do modelo deve ser definida de forma a revelar o menos possível sobre o seu funcionamento interno [Co91]. O JSD usa esse mecanismo de uma forma indireta, colocando os campos de informação das entidades em registros físicos acessados apenas por elas, isso entretanto não mascara ou impossibilita o seu uso.

  O método JSD-OO adota o encapsulamento de uma maneira mais objetiva definindo atributos e métodos internamente aos tipos-entidade. O acesso às estruturas dessas entidades, se dá apenas através de suas interfaces externas onde os métodos que manipulam essas estruturas são declarados.

- *Relacionamentos.* Uma maneira encontrada para que os modelos orientados a objetos possam gerenciar a complexidade dos sistemas foi por meio da troca de mensagens quando são estabelecidos os relacionamentos entre objetos que compoẽm o

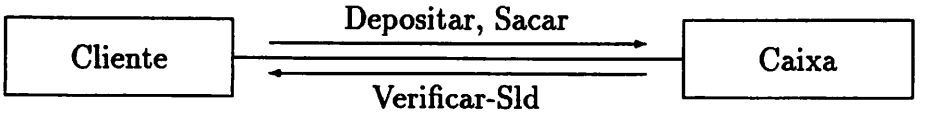


Figura 4.2: DES do Método JSD-OO

modelo, isto é, o relacionamento representa o mecanismo por onde os objetos que participam dele interagem entre si através da troca de mensagens.

No método JSD-OO vamos representar a dimensão estática do modelo pelo relacionamento entre as entidades que formam o domínio da solução através da ligação de uma linha unindo os objetos envolvidos no Diagrama de Especificação de Sistemas (DES). O nome da(s) mensagem(s) enviada(s) ou recebida(s) deve(m) ser colocada(s) logo acima da linha que une os objetos, conforme mostra o exemplo da Figura 4.2.

A distinção entre operações que modificam as estruturas de dados (*modificadoras* no OOD) e as que apenas inspecionam as estruturas sem modificá-las (*seletoras* no OOD), entre outras, devem ser caracterizadas no dicionário de dados do JSD-OO, gerado durante a construção do modelo conceitual da aplicação. Isso se deve à necessidade de adequação do modelo desenvolvido pelo JSD-OO às linguagens orientadas a objetos (LPOO) que também possuem outros tipos de operações além dessas como por exemplo, as operações de *construção* e de *destruição* de instâncias, da linguagem C^{++}.

- *Diagrama de Estruturação Cronológica.* Os modelos desenvolvidos através dos métodos de análise e projeto orientados a objetos (MAPOO's) apresentam uma visão global dos elementos que formam o domínio do problema através das várias perspectivas do sistema que existem dentro da representação do modelo de objetos. Segundo De Champeaux e Faure [Ch92], essa visão pode ser alcançada em parte por meio da dimensão dinâmica dessas perspectivas, responsável pela captura do comportamento dos objetos dentro do modelo, e pode ser representada por meio de uma máquina de estados finitos.

  O método JSD utiliza o Diagrama de Estruturação Cronológica (DEC) para representar a seqüência, ordenada em relação ao tempo, das ações que atuam sobre uma entidade. No método JSD-OO optou-se por tornar o DEC mais próximo da visão do modelo orientado a objeto de modo a poder representar também os vários estados aos quais uma entidade pode ser conduzido pela atuação de uma ação (Página 25).

  Outra modificação introduzida foi a substituição do conceito de **componentes de grupo** na notação diagramática do DEC, usada em parte para agrupar as ações que se relacionam a um determinado período da atividade da vida de uma entidade, e servindo também como delimitadores das estruturas de controle no Texto Estruturado dos processos. Para essa substituição o JSD-OO introduz elementos estruturais únicos, responsáveis por representar graficamente os tipos de estruturas

Figura 4.3: Representação das Estruturas de Controle no JSD-OO

de controle (seqüência, repetição e seleção) existentes no JSD (Figura 4.3); essas estruturas devem anteceder, no DEC modificado, o conjunto de eventos na qual elas exercem controle.

O exemplo da Figura 4.4 do sistema bancário mostra o DEC do método JSD e o DEC modificado do método JSD-OO. Nele a ação **sacar** pode levar a entidade **cliente** a dois estados diferentes (sacado e descoberto) dependendo do valor encontrado no atributo **saldo**.

Uma outra modificação adotada para o DEC foi no sentido de estruturar claramente o encapsulamento das operações de um tipo-entidade estabelecendo o local onde essas operações devam ser declaradas, formando um relacionamento de utilização entre a entidade que requer e a que sofre no caso de uma ação comum entre as duas entidades. Foram estabelecidas duas possibilidades para a definição desse local:

1. através da gramática textual fornecida pelas formas nominais do verbo como **gerúndio** para o estado de uma entidade definido por uma ação requerida de outra entidade e, **particípio** para o estado de uma entidade estabelecido pela execução de uma ação sofrida pela própria entidade;
2. através de uma forma visual, sublinhando o nome do estado alcançado pela atuação de uma ação sofrida e não sublinhando para representar o estado devido a atuação de uma ação requerida. Dessa forma a representação dos nomes dos estados pode ficar na mesma forma nominal do verbo para todas as entidades às quais a ação que determina esses estados sejam comuns (Figura 4.4). Outra observação é em relação ao posicionamento do nome da ação que determina a mudança de estado próximo ao retângulo desse estado.

Como forma de facilitar o uso de ferramentas automatizadas para desenvolvimento de sistemas utilizando o JSD-OO, adotou-se a segunda opção de representação para o DEC sendo que essa notação diagramática será mais bem visualizada através do estudo de caso do método JSD-OO apresentado ao final do capítulo.

- *Hierarquização.* Um dos principais mecanismos do modelo de objetos, senão o principal, a ser incorporado ao método JSD-OO para que a extensão do JSD possa ser

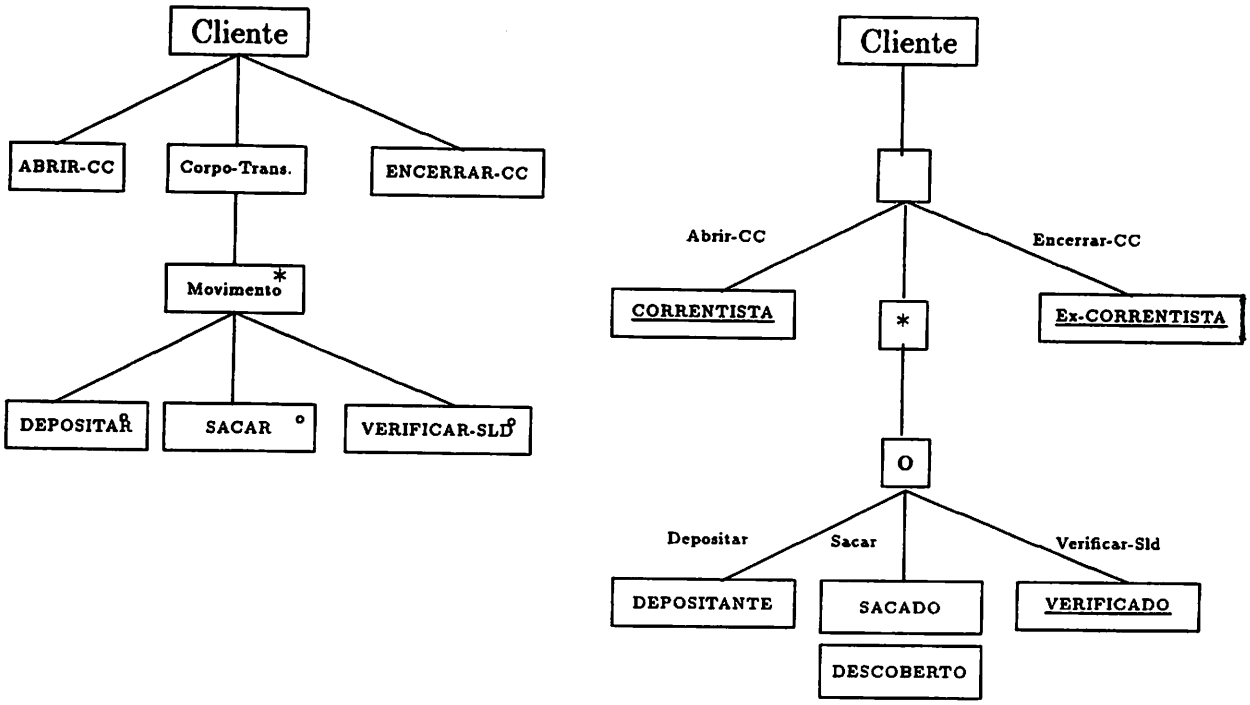


Figura 4.4: DEC para os Métodos JSD e JSD-OO

considerada orientada a objetos, é o conceito de **herança**. A herança permite que uma classe que seja subclasse de outra, herde todas as propriedades que formam a estrutura de dados da classe pai, bem como todos os procedimentos que determinam o comportamento dessa classe. O mecanismo de herança não está disponível entre os tipo-entidades no JSD, e é o motivo pelo qual não existe uma estrutura de hierarquização entre processos nesse método.

No JSD-OO será adotada a mesma idéia do formalismo apresentado em relação à herança dos diagramas de classes do método OOD. Assim, para que seja possível representar as hierarquias entre os elementos do modelo da aplicação o JSD-OO adota um retângulo de arestas descontínuas para representar o tipo-entidade e para representar os relacionamentos entre eles uma flecha com o sentido da seta apontando para o tipo-entidade da qual se herda. Na Figura 4.5, foram modelados os objetos **Cliente** e **Caixa** de modo a herdarem propriedades de uma mesma classe de objetos, a classe **Pessoa**.

- *Projeto Físico.* As quatro primeiras fases do método JSD estão associadas à modelagem conceitual e têm por função desenvolver a visão lógica do modelo. As duas últimas fases do JSD (temporização e implementação), por sua vez, estão associadas à visão física do modelo. O JSD, criado antes do desenvolvimento do paradigma de orientação a objetos [Ms88], tem no projeto físico do método, feito a partir da visão física, uma preocupação excessiva com a transformação do modelo conceitual para o ambiente computacional em que o sistema irá ser executado. Sua complexi-

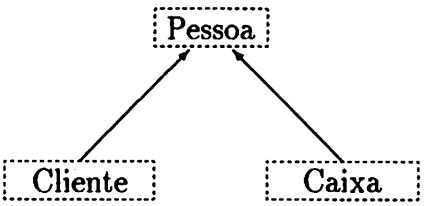


Figura 4.5: Diagrama de Herança no JSD-OO

dade se deve em grande parte à excessiva dependência do método da construção do pseudocódigo ([Ru91]).

M.A. Jackson em [Ja83] apesar de dedicar um capítulo inteiro à fase de implementação - *The Implementation Step* -, afirma logo em sua introdução:

> "*Ambientes para os quais sistemas úteis são desenvolvidos (···) fornecem algumas facilidades tais como sistemas operacionais, monitores de teleprocessamento, ou geradores de programas em lote, (···) e não existe nenhuma razão para gastar esforços recriando facilidades que já existem (···)*".

O modelo de objetos por sua vez caracteriza-se por apresentar uma pequena distinção entre as fases de projeto, lógico e físico, e entre esses e a implementação do modelo [Gi90]. Assim é que a representação da visão física de um sistema, no modelo de objetos, depende muito mais da linguagem de programação utilizada do que propriamente do método de desenvolvimento.

O JSD-OO, coerente com esse modelo, adota para representação do seu projeto físico os mesmos diagramas utilizados pelo método OOD para a representação das arquiteturas de módulos e de processos do modelo de objetos [Bo91].

O **Diagrama de Módulos - DM** é usado para mostrar a alocação de entidades e tipos-entidade em módulos no projeto físico do método JSD-OO. A Figura 4.6 mostra os símbolos utilizados pelo JSD-OO para representação do Diagrama de Módulos.

O único relacionamento incorporado entre os módulos nos diagramas de módulos é o relacionamento de dependência de compilação, que será representado no JSD-OO por uma linha com a seta apontada para o módulo pai. Os retângulos representam: 1) módulo **programa principal** que contém as chamadas para subprogramas ou outros módulos do sistema, 2) módulo **especificador das interfaces** (esqueleto) de um conjunto de operações, 3) módulo para o **corpo das operações** e que contém as implementações dos subprogramas.

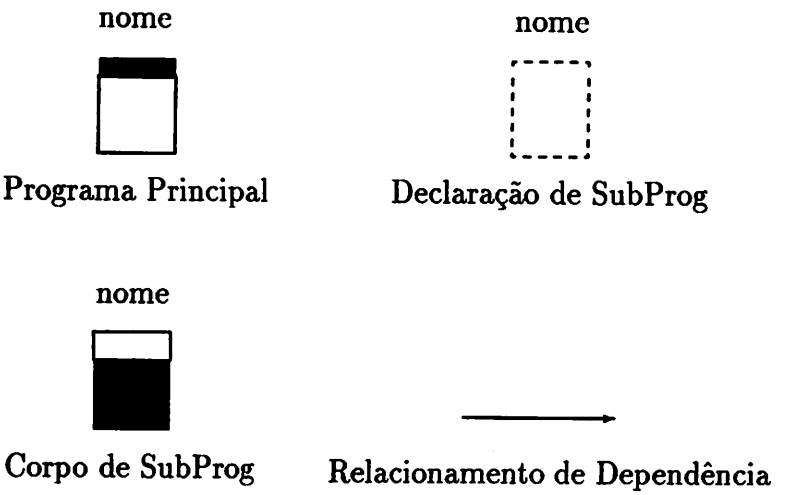


Figura 4.6: Ícones para o Diagrama de Módulos do JSD-OO

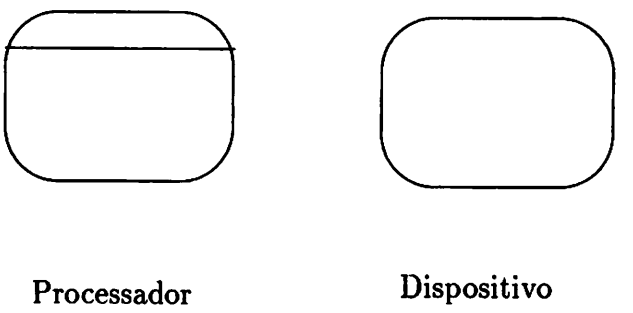


Figura 4.7: Ícones para o Diagrama de Processos do JSD-OO

O Diagrama de Processos - DP é usado para a visualização da alocação dos processos do sistema dentro dos processadores disponíveis, onde o sistema irá ser executado. No projeto físico do JSD-OO um processador do DP será representado por um retângulo de vértices arredondados e com uma secante cortando o seu topo e os dispositivos por um retângulo de vértices arredondados, como mostra a Figura 4.7.

A Tabela 4.1 apresenta uma comparação entre as características dos métodos discutidos até aqui, mostrando as alterações realizadas em relação ao JSD-OO.

Características dos métodos	JSD	OOD	JSD-OO
Elementos Chave Modeladores da Realidade	Entidades Tipos-Entidade	Objetos Classes	idem JSD
Abstração	feita a partir da realidade do problema. A abstração é realizada a partir dos elementos que formam o *universo de discurso.*	idem JSD	idem JSD
Encapsulamento	feita de modo indireto não deixando que ações de outras entidades tenham acesso ao vetor de estados de uma entidade.	Inerente aos métodos OO. Podem encapsular tanto as estruturas quanto as operações do objeto. As interfaces dos objetos são as responsáveis pela comunicação entre os objetos.	idem OOD para as entidades
Identificação	entidades identificadas de modo único através dos atributos chave.	objetos identificados de modo único dependendo da LPOO utilizada quando da sua instanciação.	idem OOD para as entidades
Relacionamento	feita através de conexões entre entidades. A conexão por mensagem, que permite uma visibilidade apenas parcial dos dados e a conexão por inspeção que dá uma visibilidade total dos dados de uma entidade. Não existe o relacionamento de herança.	dependente da LPOO usada para a construção do sistema. Os relacionamentos podem acontecer entre classes e entre objetos. O relacionamento básico é o de herança.	idem OOD; eliminada a preocupação do JSD em caracterizar o tipo de conexão no DES.
Formalismos para representação diagramática			
Dimensão Estática	DES	DO,DC	continua a utilizar o DES eliminando apenas a representação do tipo de conexão, acrescentando as mensagens enviadas ou recebidas por uma entidade.
Dimensão dinâmica	DEC	DTE,DT	incorpora ao DEC modificado a representação dos estados das entidades devido à mudança das ações.
Visão Física	DIS	DM,DP	adota a visão física do OOD incorporando os diagramas de módulos e de processos.

Tabela 4.1: Comparação entre os Métodos

4.2 Estudo de Caso

Para que se possa fazer uma melhor avaliação do método, a seguir será apresentado um estudo de caso utilizando o método JSD-OO. O enunciado do problema foi retirado do livro [Ms92], sendo a solução dada originalmente utilizando-se o método JSD (Apêndice C).

O objetivo desse estudo de caso é o de mostrar apenas a viabilidade de utilização do JSD-OO como método de modelagem de sistemas de informação sobre o paradigma de orientação a objetos, não sendo levado em consideração detalhes como tipo e tamanho dos atributos das entidades.

4.2.1 O Problema

A Srta. Doralice Sampaio é a bibliotecária chefe da Biblioteca do Instituto de Matemática e Computação (IMC) de conhecida universidade brasileira. Ultimamente, esse Instituto vem recebendo muitos microcomputadores para uso de seus alunos e, como há tantos disponíveis, o diretor da escola acenou com a possibilidade de instalar um deles na biblioteca.

A biblioteca possui ainda uma funcionária para comprar livros das editoras ou livrarias. Quando o livro é recebido pela biblioteca ele é tombado, classificado por assunto e catalogado, sendo em seguida liberado para empréstimo ou consulta. O acervo fica aberto para que os leitores possam ir até às estantes e escolher o livro desejado. Ao sair, uma atendente anota o empréstimo.

Os professores podem indicar livros para serem comprados. As indicações são mantidas até que haja verba disponível para os livros indicados serem comprados. Os livros com maior número de indicações têm prioridade, mas pode ocorrer que livros não indicados formalmente sejam comprados também. A biblioteca aceita doações de livros e quando algum leitor perde um livro emprestado, ele deve repor o livro.

Os empréstimos podem ser renovados uma quantidade ilimitada de vezes, a menos de alguns livros muito consultados ou então quando ocorrer uma reserva. Livros usados como texto básico em cursos do IMC, durante o período letivo, não podem ser emprestados, devendo ser consultados nas dependências da biblioteca.

Os leitores devem se cadastrar como usuários da biblioteca, e só podem ser alunos, ex-alunos, professores e funcionários. Os leitores têm um prazo fixo de tempo para empréstimo e um limite de livros emprestados simultaneamente, com exceção dos professores, que não têm limite na quantidade de livros emprestados.

O leitor pode desligar-se espontaneamente da biblioteca, cancelando a sua inscrição, ou pode ser desligado à revelia depois de um período longo sem utilizar a biblioteca. Isso ocorre com freqüência com os alunos que se formam e se mudam da cidade.

4.2.2 Escolha das Entidades e Ações

Dicionário de Dados

entidade: Indicação
entidade pai: não tem
ações requeridas: comprar
ações sofridas: indicar, desistir, receber
estados possíveis: indicado, reindicado, desistência, desistência final, comprando e recebido
atributos: n_indic, título, autor, editora, n_edic, data_publ, data_indic, leitor_indic, data_aquis, preço

entidade: Livro
entidade pai: não tem
ações requeridas: receber
ações sofridas: comprar, repor, doar, classificar, liberar com renovação, liberar sem renovação, emprestar, emprestar sob reserva, renovar, devolver, reter para consulta, retirar, retornar, perder, expurgar
estados possíveis: comprado, reposto, doado, classificado, liberado com, liberado sem, emprestado, emprestado sob reserva, renovado, devolvido, retido, retirado, retornado, perdido, expurgado, recebendo
atributos: n_livro, n_indic, data_aquis, preço, título, autor, editora, n_edic, data_publ, classificação, data_empr, data_devol, período_ret, data_perda, data_expur, motivo_expur

entidade: Leitor
entidade pai: não tem
ações requeridas: repor, emprestar, renovar, devolver, retirar, retornar, reservar, cancelar
ações sofridas: inscrever, atualizar dados, desligar
estados possíveis: inscrito, atualizado, desligado, repondo, emprestando, renovando, devolvendo, retirando, retornando, reservando e cancelando
atributos: n_insc, nome, categoria, matrícula, identidade, ender, validade e data_insc

entidade: Professor
entidade pai: Leitor
ações requeridas: indicar, desistir
ações sofridas: idem leitor
estados possíveis: indicando, desistindo
atributos: idem Leitor

entidade: Aluno
entidade pai: Leitor

ações requeridas: idem entidade pai
ações sofridas: idem entidade pai
estados possíveis: idem entidade pai
atributos: idem entidade pai

entidade: Reserva
entidade pai: não tem
ações requeridas: emprestar sob reserva
ações sofridas: reservar, cancelar
estados possíveis: emprestando sob reserva, reservado, cancelado
atributos: n_reserva, n_leitor, n_livro, data_reserva

Todas as ações definidas pelas entidades até agora alteram de alguma forma suas estruturas de dados e portanto são consideradas como operações modificadoras de estrutura. Sendo assim não há necessidade de distinção entre elas, como seria o caso se houvesse alguma ação que apenas verificasse o estado das variáveis da entidade sem entretanto fazer alterações.

4.2.3 Estrutração Cronológica e Hierarquização

Diagramas de Estruturação Cronológica

Algumas alterações na análise e especificação dos requisitos do problema foram feitas em relação a solução JSD de [Ms92]. Isso se deve à visão que o JSD-OO oferece no domínio da solução, quando utilizando suas técnicas de desenvolvimento. No Diagrama de Estruturação Cronológica da entidade *Indicação* foi eliminado o uso da ação *Reindicar*, pois seu efeito e atuação produzem os mesmos resultados que o uso da ação *Indicar* também modelada. A situação da entidade *Indicação* apenas altera o estado quando uma instância qualquer de *Livro* é indicada pela segunda vez, devendo a entidade *Indicação* passar do estado de *Indicado* para o estado de *Reindicado* (Figura 4.8).

Outra mudança efetuada foi em relação a atuação da ação *Encomendar* da mesma entidade *Indicação*, que tem a função de armazenar dados como: nome do vendedor, data da encomenda, preço, etc. Em paralelo à essa ação, o modelo define na entidade *Livro* a ação *Comprar* que atribui dados de livros do tipo: data da aquisição, quantidade recebida, etc. O modelo JSD-OO coloca as atribuições das duas ações como parte do método *Comprar* definido no tipo-entidade *Livro* e estabelece um relacionamento entre instâncias de *Indicação* e de *Livro* na qual mensagens são enviadas do primeiro para o segundo.

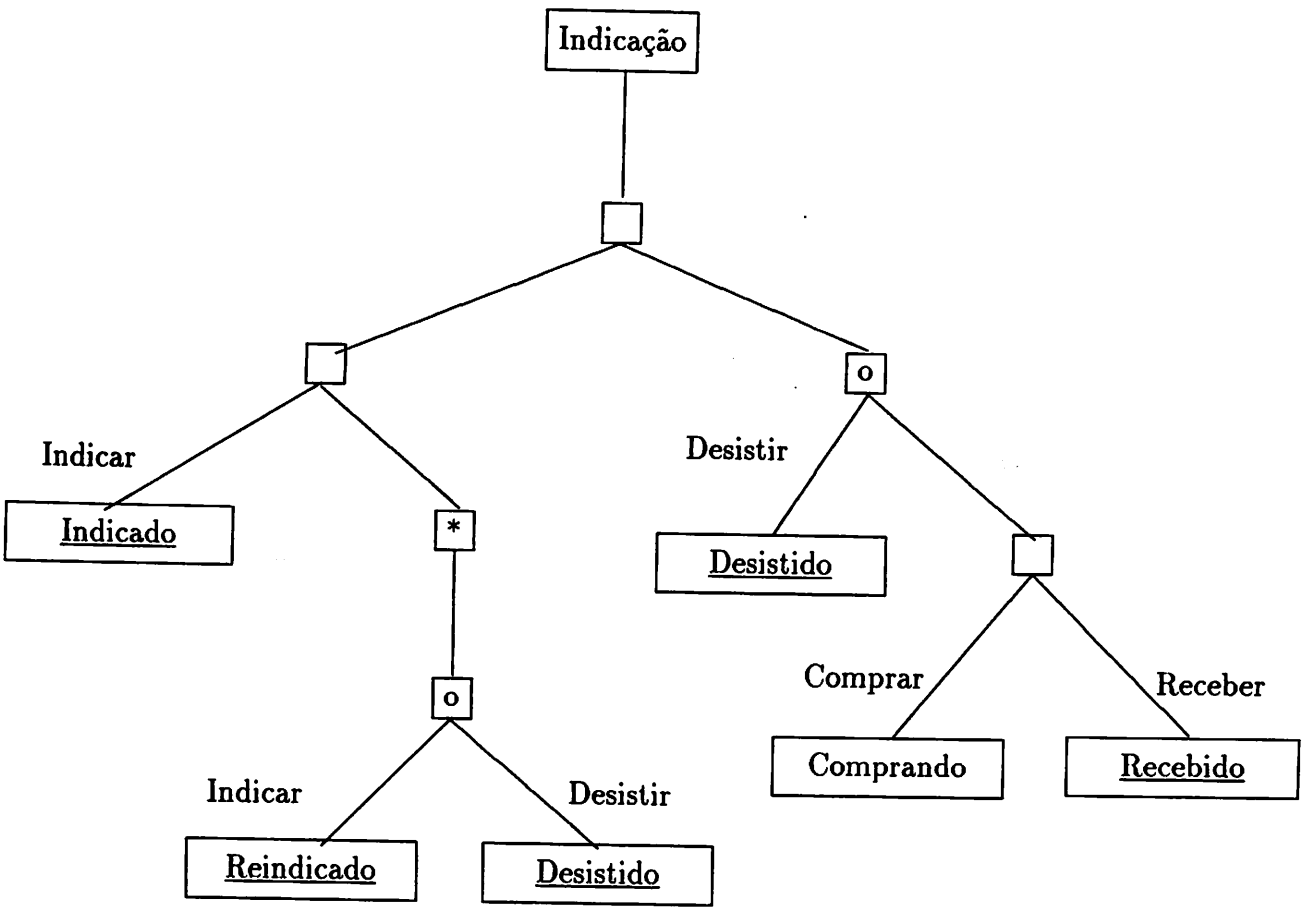
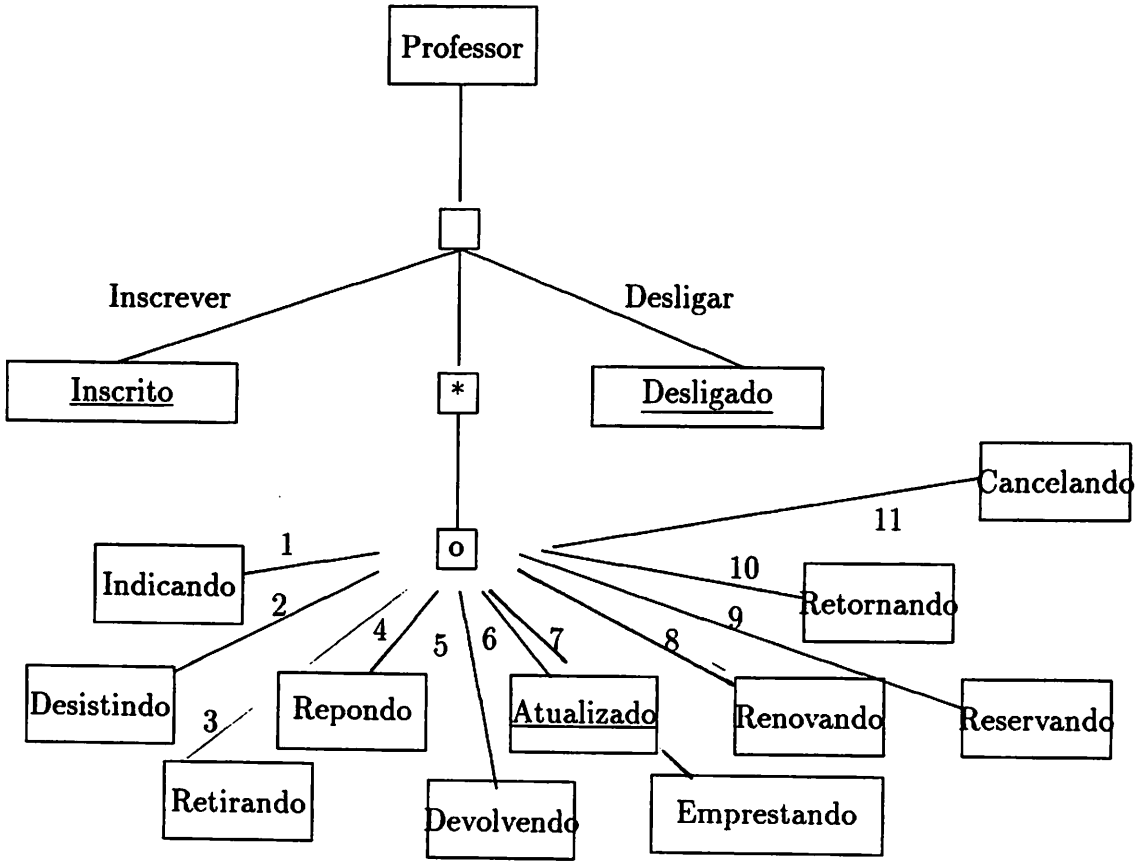


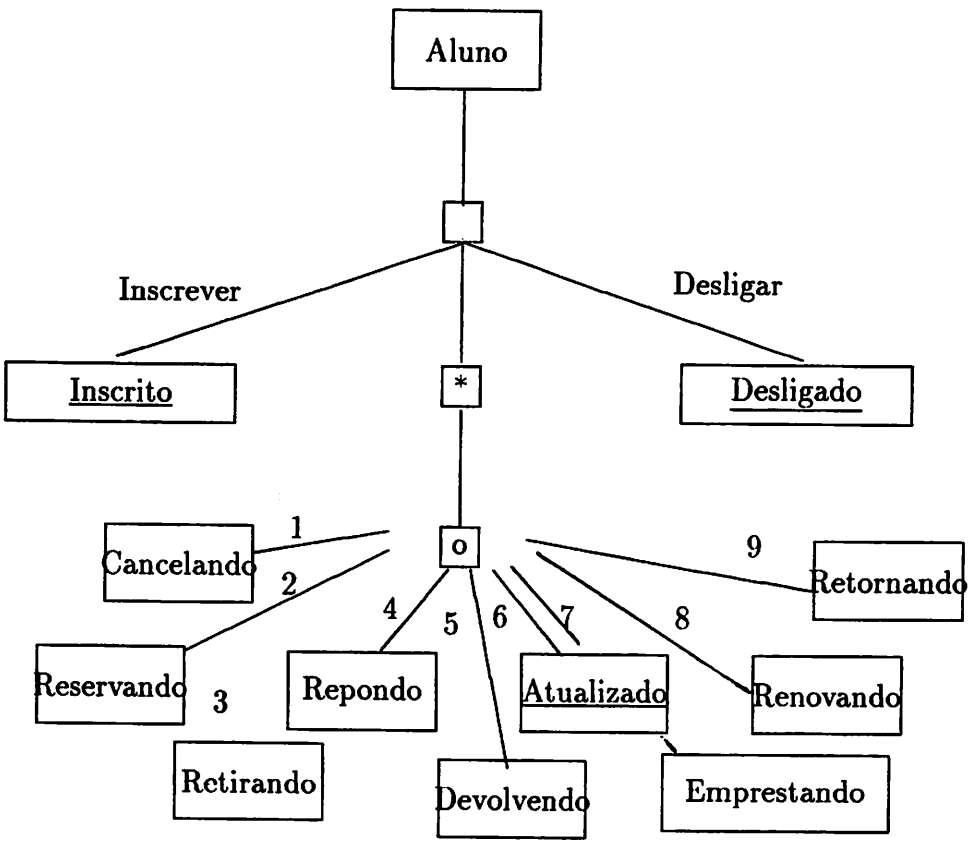
Figura 4.8: DEC da Entidade Indicação



Legenda de Ações:

1 - Indicar	6 - Atualizar Dados
2 - Desistir	7 - Emprestar
3 - Retirar	8 - Renovar
4 - Repor	9 - Reservar
5 - Devolver	10 - Retornar
	11 - Cancelar

Figura 4.9: DEC da Entidade Professor



Legenda de Ações:

1 - Cancelar
2 - Reservar
3 - Retirar
4 - Repor
5 - Devolver
6 - Atualizar Dados
7 - Emprestar
8 - Renovar
9 - Retornar

Figura 4.10: DEC da Entidade Aluno

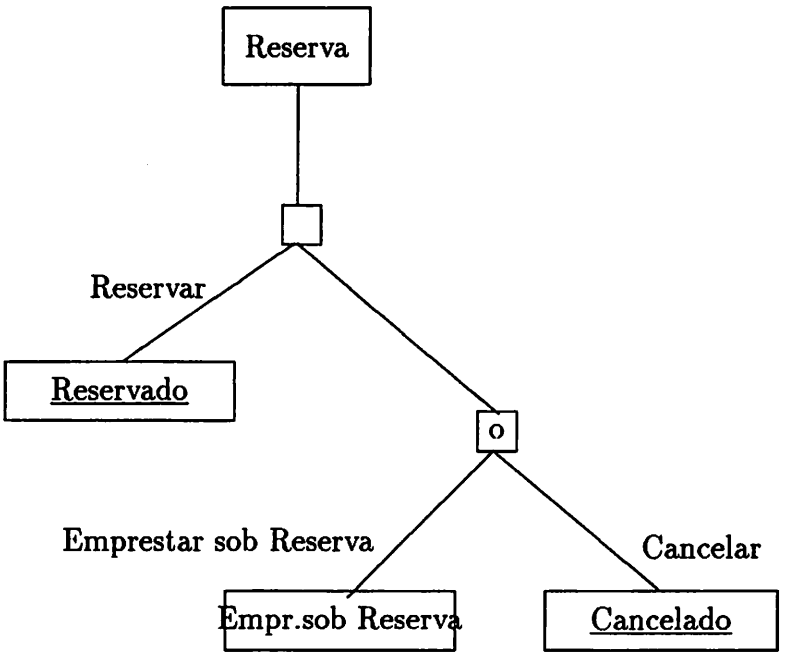


Figura 4.11: DEC da Entidade Reserva

O Diagrama de Estruturas que se segue (Figura 4.12) reflete as mudanças efetuadas no modelo de desenvolvimento quando da análise do domínio do problema sob a visão do método JSD-OO. Ele representa os estados possíveis da entidade *Livro* devido à atuação de eventos tratados pela própria entidade (no dicionário de dados os estados que se encontram na forma nominal do verbo no particípio) e os estados possíveis devido à atuação de ações requeridas pela entidade tratadas em outro tipo-entidade (no dicionário de dados os estados que se encontram na forma nominal do verbo no gerúndio). Essas mudanças são determinantes de uma melhor formalização da visão dinâmica dos eventos que atuam sobre as entidades no domínio da aplicação e são apresentadas a seguir:

- apresenta a atuação da ação *Receber* sobre a entidade por um retângulo sem uma linha no canto superior direito indicando que essa ação é tratada em outro tipo-entidade, no caso no tipo-entidade *Indicação*;
- inclui a representação gráfica da ação *Emprestar sob Reserva* (retângulo com vértice superior direito cortado) como sendo definida na própria entidade através da declaracão procedimental que trata do evento.

Uma adaptação necessária feita no DEC modificado da entidade *Livro*, foi a inclusão da possibilidade de se continuar no mesmo estado devido à escolha de uma opção numa estrutura de controle de Seleção. No diagrama, quando um Livro é liberado para empréstimo sem direito a renovação, significando que leitor não pode tomar emprestado o livro mais de uma vez, essa opção é representada colocando-se a palavra **(nenhuma)** entre parentêses

junto à linha que une a estrutura de seleção ao estado da entidade *Livro*, determinando uma ação (ou ausência de ação) a ser executada. Essa ausência de ação faz com que a entidade permaneça no mesmo estado em que se encontrava anteriormente. Uma observação nesse sentido é o fato de que os estados porque passa uma entidade podem ser acumulativos, isto é, a permanência num estado devido a atuação de uma ação não acaba necessariamente com a permanência da entidade em outro estado anterior. Como exemplo dessa situação a entidade *Livro* pode ficar no estado de *Liberado com Renovação* e num instante posterior, de modo acumulativo, passar para o estado de *Emprestado.*

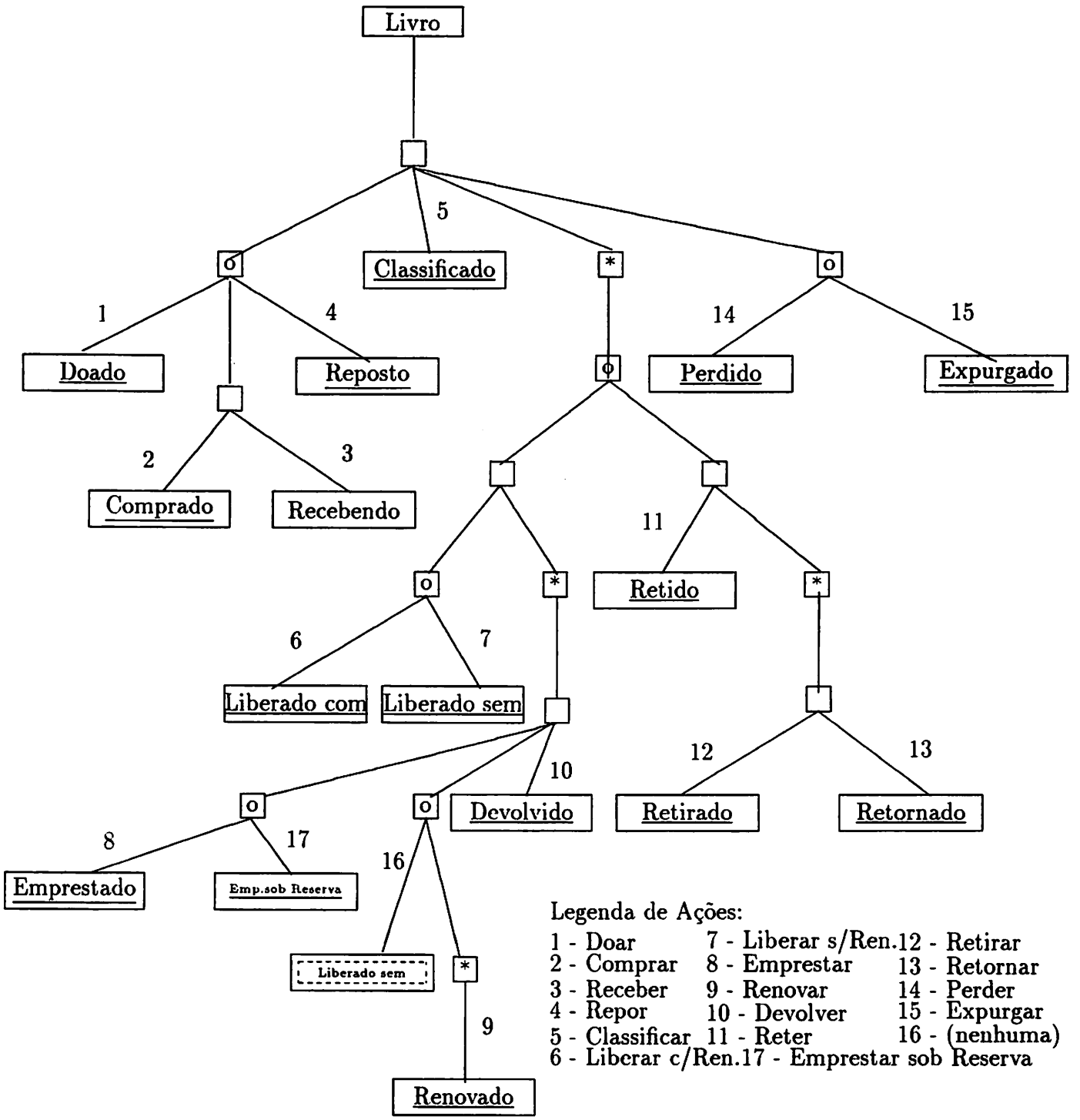


Figura 4.12: DEC da Entidade Livro

Diagrama de Herança

Como visto na apresentação do método JSD-OO, Capítulo 4, a herança é o principal fator introduzido pelo método para que se possa conceituar o modelo desenvolvido como sendo o estabelecido pelo paradigma de orientação a objetos. No estudo de caso, a entidade *Leitor* torna-se, dentre as modificações efetuadas, uma abstração de mais alto nível das entidades *Professor* e *Aluno* (Figura 4.13), isto é, os procedimentos similares às duas últimas entidades são definidos apenas em *Leitor*. Em *Professor* são definidos apenas os procedimentos para o tratamento dos eventos provocados por uma indicação de livros à biblioteca e que só podem ser feitas por instâncias desse tipo-entidade. Mais uma vez enfatizamos a equivalência semântica entre herança nos métodos **OO** e o no método JSD-OO, através do reaproveitamento estrutural e funcional de tipo-entidades mais abstratas por seus descendentes.

Os diagramas DEC das entidades *Professor* e *Aluno* mostrados nas Figuras 4.9 e 4.10 respectivamente, detalham os aspectos de seqüenciamento e estados possíveis devido à atuação dos eventos que atuam sobre elas; entretanto isso não significa que os procedimentos que tratam desses eventos sejam definidos nos respectivos tipos-entidade, eles podem ter sido definidos em outras abstrações de mais alto nível (*Leitor*).

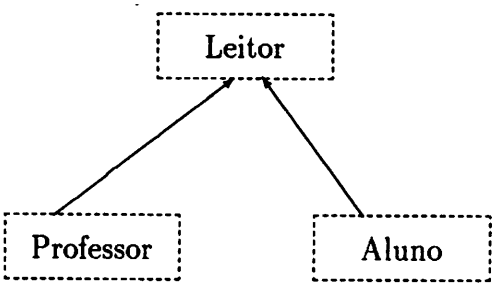


Figura 4.13: Diagrama de Herança

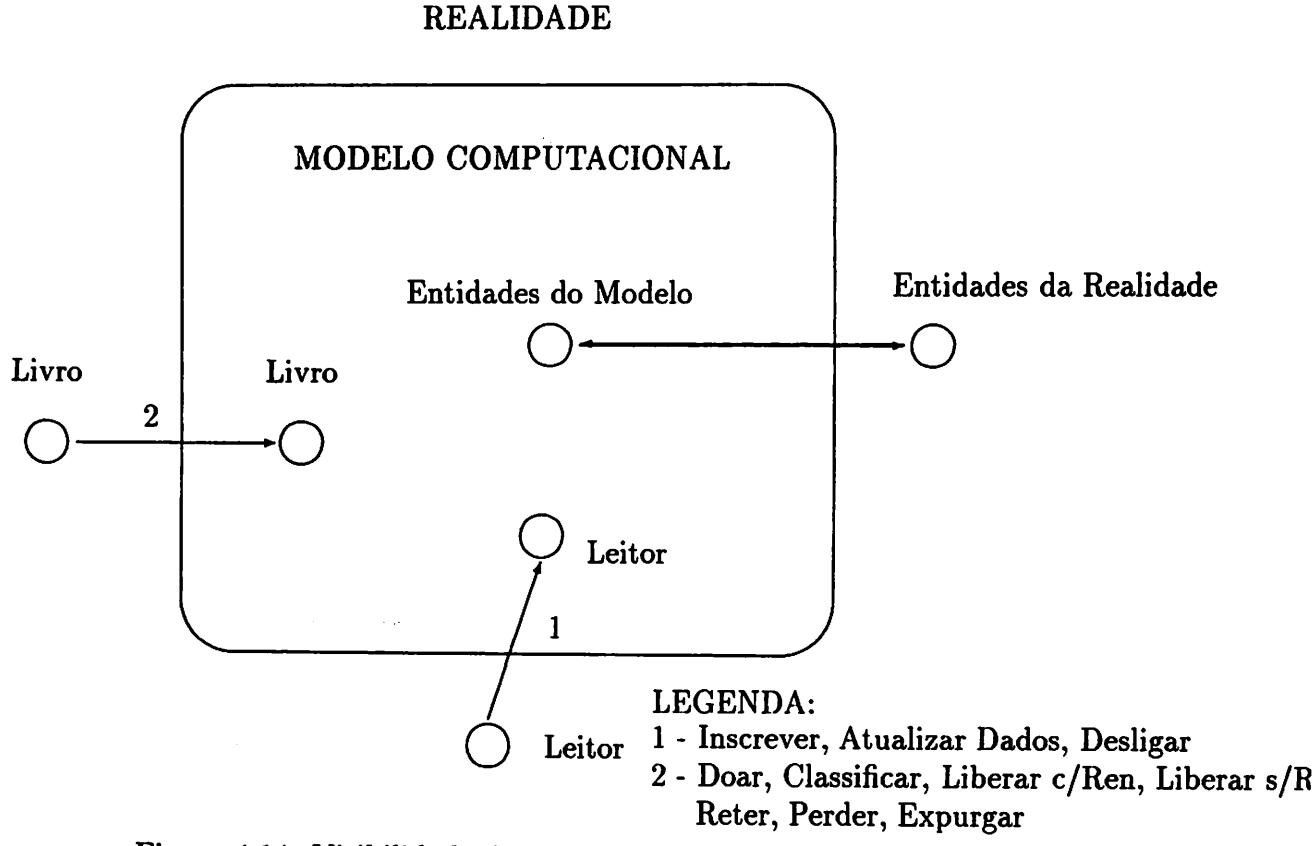


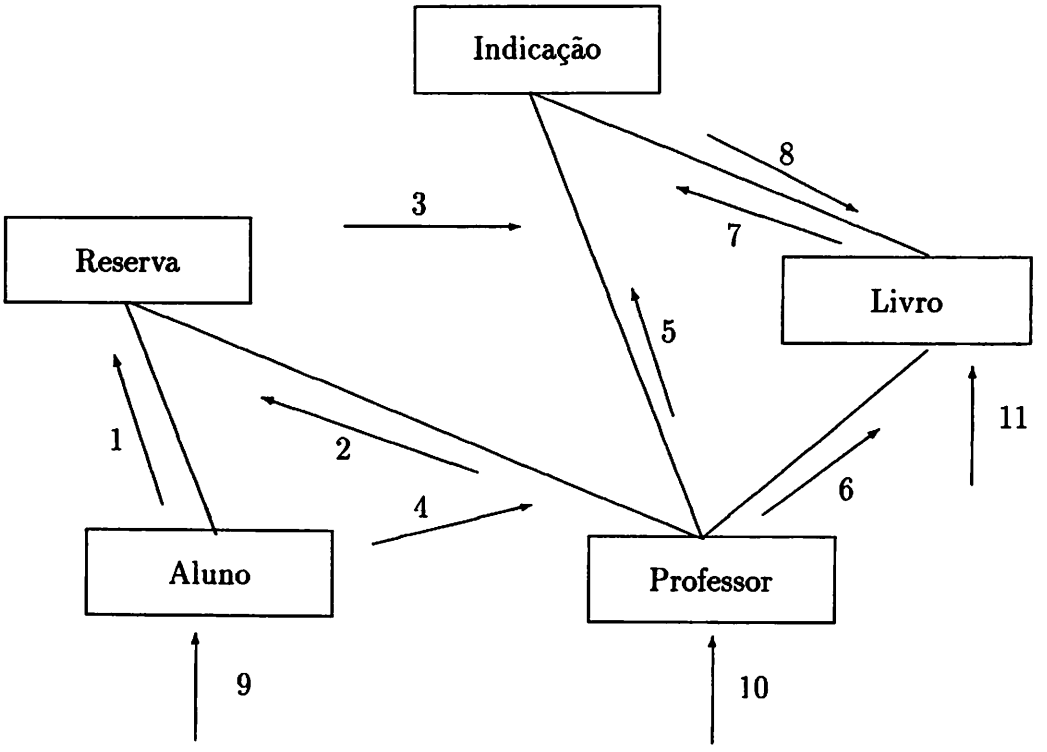
Figura 4.14: Visibilidade das Operações Além Fronteiras

4.2.4 Rede de Processos

Diagrama de Especificação de Sistemas

O Diagrama de Especificação de Sistemas (DES) do modelo JSD-OO representa o relacionamento existente entre as entidades que se interagem a fim de que o sistema como um todo reproduza computacionalmente a realidade do domínio do problema. O DES modificado, do JSD-OO, é comparável ao Diagrama de Objetos do modelo objeto.

A Figura 4.15 mostra a visualização das mensagens enviadas ou recebidas por uma entidade, numeradas de acordo com o conteúdo da legenda. Os rótulos 9, 10 e 11 correspondem às mensagens enviadas por objetos reais externos às fronteiras do sistema e recebidas e tratadas por entidades modeladas pelo uso do método e que de uma certa maneira simulam a sua existência. O que caracteriza essas operações quanto à utilização das outras, é o âmbito de atuação de seus comandos, exclusivos das entidades que recebem as mensagens, não existindo por parte de outras entidades do modelo uma visibilidade em relação a elas. Pode-se dizer que esse tipo de operação "enxerga" apenas as entidades do mundo real, fora das fronteiras do sistema (Figura 4.14).



LEGENDA:

1-Reservar, Cancelar
2-idem 1
3-Emprestar sob Reserva
4-Repor, Emprestar, Renovar, Devolver,Retirar, Retornar
5-Indicar, Desistir
6-idem 4
7-Receber
8-Comprar
9-Inscrever, Atualizar Dados, Desligar
10-Doar, Classificar, Liberar c/Ren., L Reter, Perder, Expurgar

Figura 4.15: Diagrama de Especificação de Sistemas

Funções

O modelo JSD-OO até aqui desenvolvido apresenta as entidades como sendo uma extensão do mundo real, simulando as atividades reais do meio externo. Entretanto o modelo JSD-OO para ser útil deve fornecer meios para que os resultados, derivados da atividade de processamento dos dados de entrada fornecidos pelo mundo real, sejam expostos novamente de volta ao mundo real através de relatórios, por exemplo. O modelo JSD, como já visto, nesta etapa incorpora novos processos (no caso das funções impostas) ou procedimentos ao modelo inicial, sem que haja qualquer tipo de mudança estrutural significativa no modelo anterior. No modelo JSD-OO o tratamento dos dados de entrada pode em alguns casos, ser realizado pelas próprias entidades responsáveis e novas operações podem ser definidas nestas entidades com o objetivo de fornecerem as saídas do sistema.

Neste estudo de caso serão mostrados dois mecanismos para a inclusão de procedimentos ao modelo JSD-OO: (1) adicionar uma nova entidade[1] ao modelo para realizar uma atividade de informação e (2) adicionar um procedimento a uma entidade existente. Os exemplos abaixo das funções retiradas de [Ms92] exemplificam o processo de uso desses mecanismos ao serem incorporados ao modelo.

- Suponha que a bibliotecária solicite um relatório de todos os livros comprados até aquele instante do ano e o custo total para a aquisição.

  Para que esta função possa ser implementada o projetista deve definir uma nova estrutura de dados onde os dados referentes a totalização dos livros e seus custos sejam armazenados (tot_livros, tot_custos). Uma nova entidade, *Biblioteca*, deve ser criada artificialmente encapsulando em sua estrutura os novos dados e que implemente a ação *Listar* que verifica todas as instâncias de *Livro* adquiridas até aquele instante, produzindo o relatório solicitado. Esse procedimento deverá: (1) especificar um método de acesso a todas as instâncias da entidade *Livro* procurando pelas que tenha o valor da variável *data_aquis* maior que o parâmetro de entrada, (2) totalizar a quantidade e o valor de cada aquisição e (3) listar as variáveis *n_livro*, *título*, *preço* e *data_aquis* de cada uma das instâncias que se encaixem ao pedido. Para conseguir implementar um procedimento assim a entidade *Biblioteca* pode ser idealizada como uma composição de todas as instâncias da entidade *Livro*.

  As mudanças efetuadas devem ser transmitidas ao modelo pela geração dos novos diagramas de estruturas, atualização do DES incorporando os novos relacionamentos (ação *Listar* entre Livro e Biblioteca, e ação *Verificar_Data* entre Biblioteca e Livro) e, por último, atualizando o Dicionário de Dados que deve estar permanentemente atualizado para reproduzir as situações e estados do modelo que expressem a realidade.

  O DEC da entidade *Biblioteca* mostra as ações *Listar* e *Verificar_Data* sendo exe-

[1] Na verdade uma Pseudo-Entidade já que ela é criada artificialmente e não corresponde a rigor a nenhuma entidade do domínio do problema.

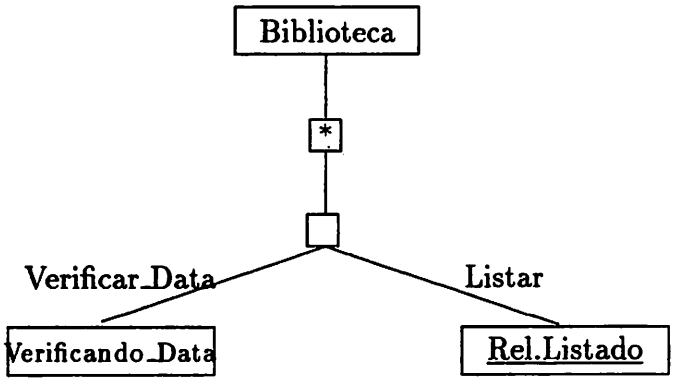


Figura 4.16: DEC da Entidade Biblioteca

cutadas iterativamente (Figura 4.16). A Figura 4.17 mostra o Diagrama de Especificação de Sistemas com a nova entidade. O Dicionário de Dados referente às entidades alteradas, atualizada segundo essa nova visão do modelo é visto abaixo. A ação *Verificar_Data* deve ser sublinhada para diferenciar uma operação modificadora (as sem sublinhado) daquelas operações chamadas seletoras (as sublinhadas) que apenas inspecionam as variáveis sem alterar os seus valores.

entidade: Livro
entidade pai: não tem
ações requeridas: receber
ações sofridas: comprar, repor, doar, classificar, liberar com renovação, liberar sem renovação, emprestar, emprestar sob reserva, renovar, devolver, reter para consulta, retirar, retornar, perder, expurgar, verificar_data
estados possíveis: comprado, reposto, doado, classificado, liberado com, liberado sem, emprestado, emprestado sob reserva, renovado, devolvido, retido, retirado, retornado, perdido, expurgado, recebendo, data_checada
atributos: n_livro, n_indic, data_aquis, preço, título, autor, editora, n_edic, data_publ, classificação, data_empr, data_devol, período_ret, data_perda, data_expur, motivo_expur

entidade: Biblioteca
entidade pai: não tem
ações requeridas: verificar_data
ações sofridas: listar
estados possíveis: verificando_data, rel_listado
atributos: tot_custos, tot_livros

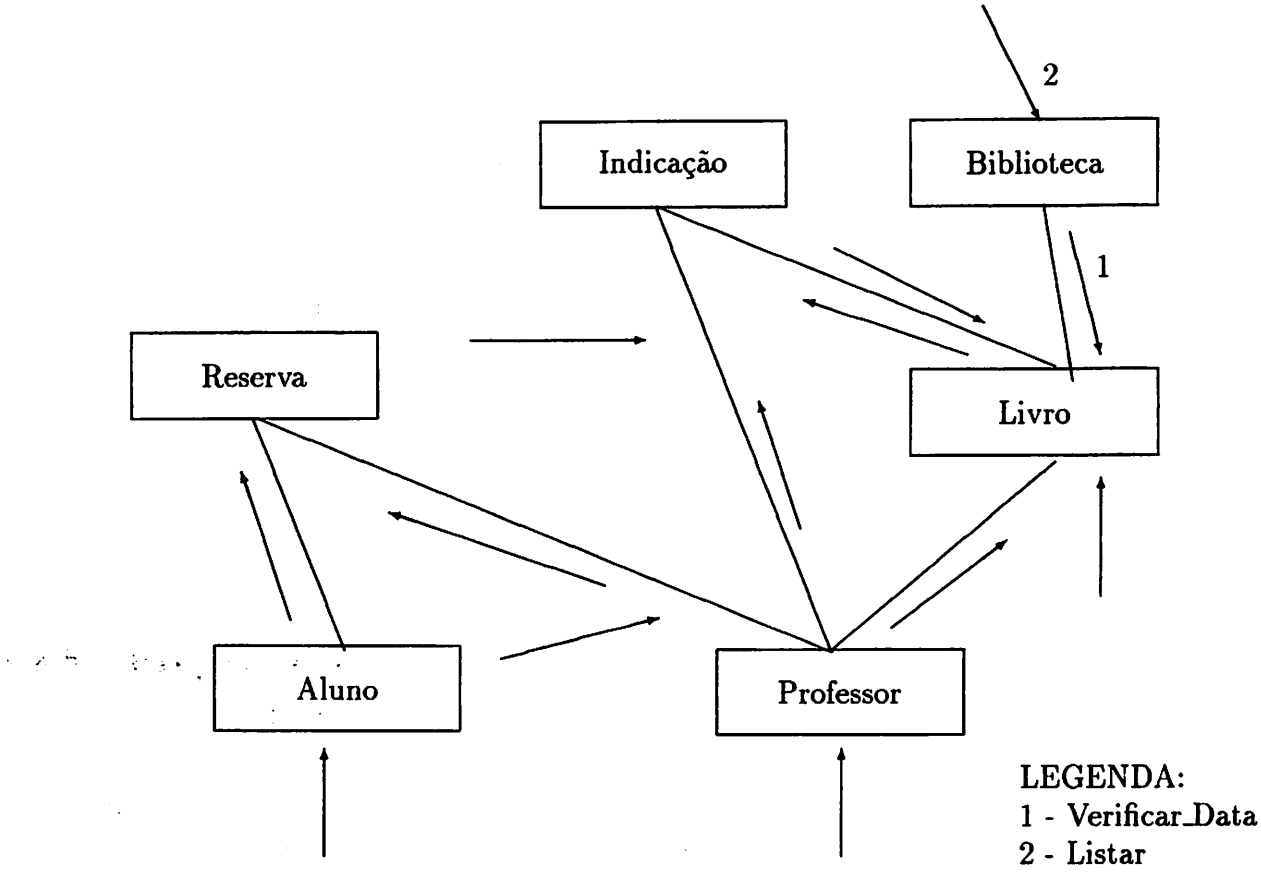


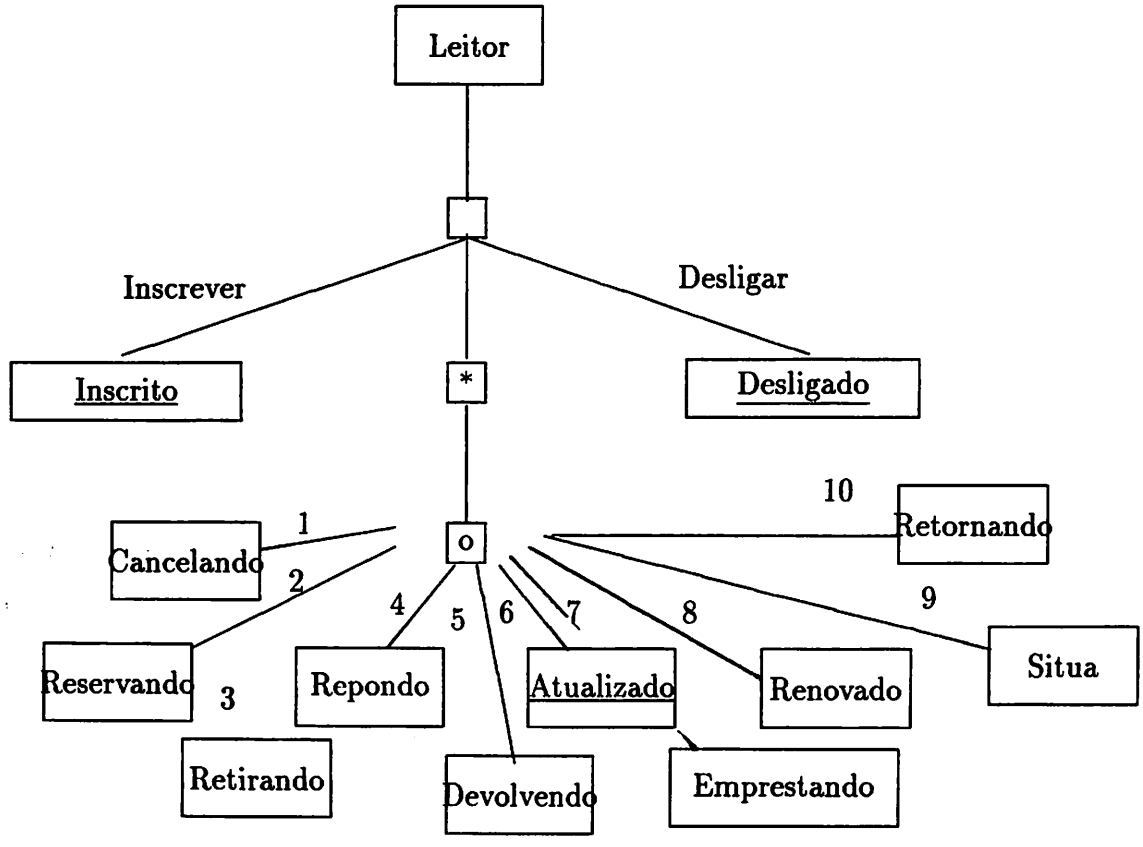
Figura 4.17: DES destacando a Entidade Biblioteca

- Dada a identificação de um livro, indicar sua situação, isto é, se está na estante ou emprestado e, nesse caso para quem.

  Esta função, executada sob demanda, fornece a situação de um livro específico, isto é, se ele encontra-se na estante ou se emprestado, e nesse caso para quem. Sua inclusão no modelo não necessita da criação de uma nova entidade pois ela é fruto do relacionamento básico entre as entidades *Leitor* e *Livro*. Como observado no método JSD, uma função com essas características é definida como sendo embutida, pois o texto de sua execução fica junto ao texto estruturado do processo da entidade. A solução JSD-OO também permite que se coloque essa solicitação junto a uma entidade qualquer (por exemplo na entidade *Livro*) a responsabilidade pela sua execução definindo-a como uma ação sofrida, e encapsulando-a juntamente com as outras operações e estruturas da entidade.

  A operação que realiza a consulta foi designada como *Situar* e a Figura 4.18 mostra parte do Diagrama de Estruturas da entidade *Leitor* apresentando a nova ação requerida.

  As modificações necessárias para atender o novo requisito são: (1) acréscimo do atributo lógico *situação* à entidade *Livro* que armazena a situação do livro (True



Legenda de Ações:
1 - Cancelar 6 - Atualizar Dados
2 - Reservar 7 - Emprestar
3 - Retirar 8 - Renovar
4 - Repor 9 - Situar
5 - Devolver 10 - Retornar

Figura 4.18: DEC de *Leitor* com a ação Situar

para livro na estante e False caso contrário) e (2) inclusão da operação *Situar* junto à entidade *Livro*. A Figura 4.19 mostra o Diagrama de Especificação com a nova operação *situar* representada. O Diagrama de Especificação de Sistemas mostra a nova ligação entre as entidades *Leitor* e *Livro*, indicando que uma nova comunicação entre elas foi estabelecida.

O Dicionário de Dados deve ser atualizado paralelamente às alterações efetuadas nas entidades *Livro*, *Professor* e *Aluno* expressando um estado de consistência entre o DD e os diagramas. A alteração realizada nas entidades *Professor* e *Aluno* se processa apenas na entidade-pai *Leitor*, pois a operação *situar* é herdada por ambas:

entidade: Livro
entidade pai: não tem
ações requeridas: receber
ações sofridas: comprar, repor, doar, classificar, liberar com renovação, liberar sem renovação, emprestar, emprestar sob reserva, renovar, devolver, reter para consulta, retirar, retornar, perder, expurgar, verificar_data, situar
estados possíveis: comprado, reposto, doado, classificado, liberado com, liberado sem, emprestado, emprestado sob reserva, renovado, devolvido, retido, retirado, retornado, perdido, expurgado, recebendo, data_verificada, situado
atributos: n_livro, n_indic, data_aquis, preço, título, autor, editora, n_edic, data_publ, classificação, data_empr, data_devol, período_ret, data_perda, data_expur, motivo_expur, situação

entidade: Leitor
entidade pai: não tem
ações requeridas: repor, emprestar, renovar, devolver, retirar, retornar, reservar, cancelar, situar
ações sofridas: inscrever, atualizar dados, desligar
estados possíveis: inscrito, atualizado, desligado, repondo, emprestando, renovando, devolvendo, retirando, retornando, reservando, cancelando e situado
atributos: n_insc, nome, categoria, matrícula, identidade, ender, validade e data_insc

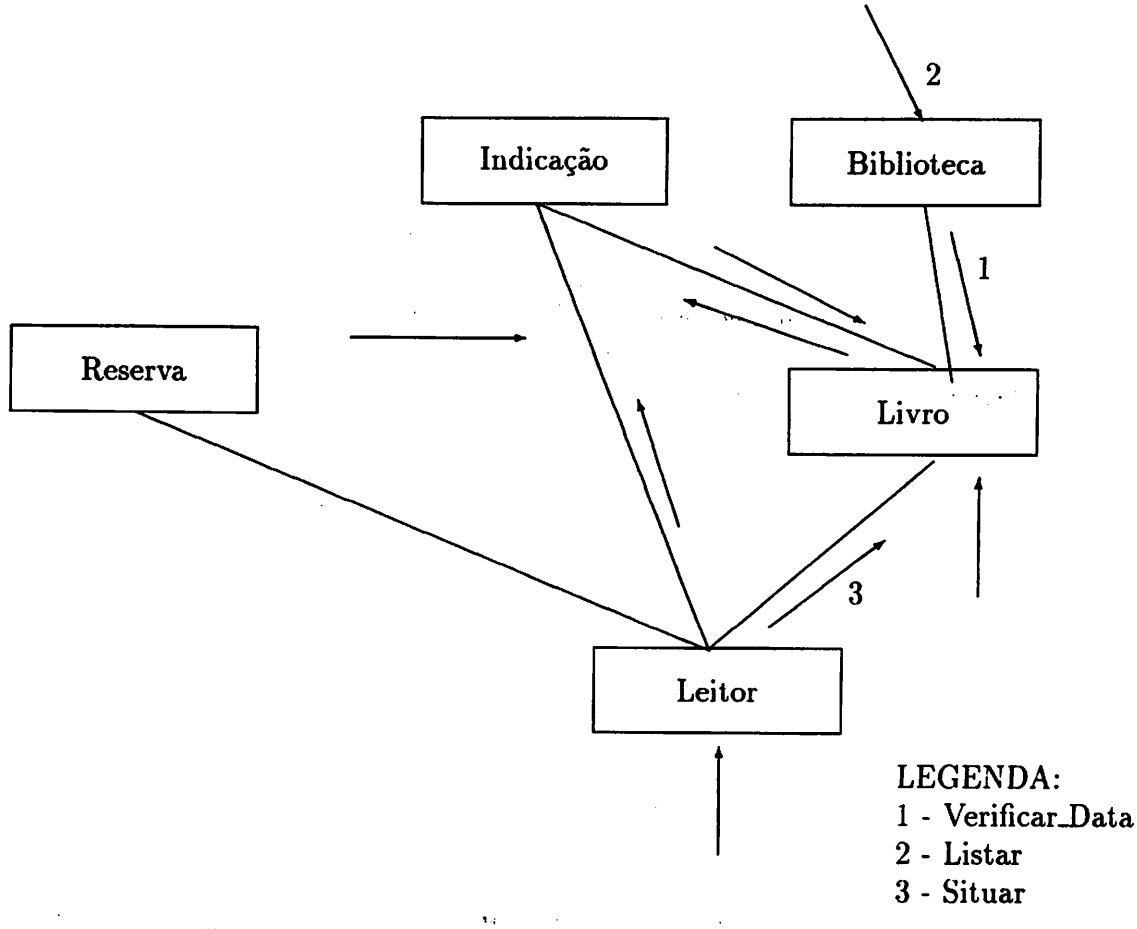


Figura 4.19: DES acrescido da ação Situar

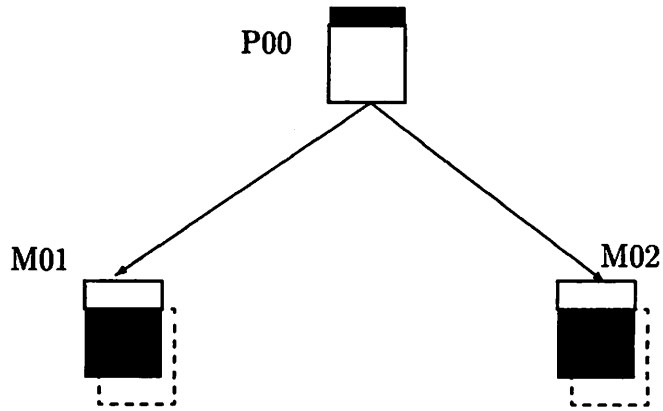


Figura 4.20: DM do Sistema de Biblioteca

4.2.5 Projeto Físico

Diagrama de Módulos

Em grandes sistemas, com centenas de classes, o uso de módulos se torna essencial para auxílio no gerenciamento da complexidade [Bo91]. Os módulos são o local onde tipos-entidade relacionados são colocados, para formarem parte da arquitetura física do modelo JSD-OO (A outra parte da arquitetura física é determinada pela Arquitetura de Processos).

Nesse exemplo, tratando-se de poucos, todos os tipos-entidade poderiam ter sidos colocados num único módulo separando apenas a interface do módulo da sua implementação por questões de manutenção. O modelo objeto, entretanto, prevê a possibilidade de reutilização de classes de um sistema em outros. Assim as classes mais relacionadas, aquelas que dizem respeito a um mesmo tipo de atuação com pequenas mudanças de comportamento, podem ser colocadas num mesmo módulo, facilitando o gerenciamento de classes com esse intuito. No exemplo os tipos-entidade Leitor, Professor e Aluno estão relacionados e por isso são colocados num mesmo módulo; Livro, Indicação, Reserva e Biblioteca em um outro módulo.

Vendo em mais detalhes a modulazição do exemplo, observa-se uma dependência estrutural de *leitor* em relação ao tipo-entidade *livro* (variável *situação*). Booch em [Bo91], afirma que esse tipo de dependência só deve existir entre classes que pertençam a um mesmo módulo. Uma vez mais se faz necessário fazer uma alteração na modularização, colocando-se o tipo-entidade *livro* no mesmo módulo que o tipo-entidade (e seus subtipo-entidades) *leitor*. A Figura 4.20 mostra o Diagrama de Módulos do modelo JSD-OO. O programa principal (P00) é o responsável pela chamada dos módulos e uma relação de dependência de compilação existe entre ele e os módulos M01 e M02.

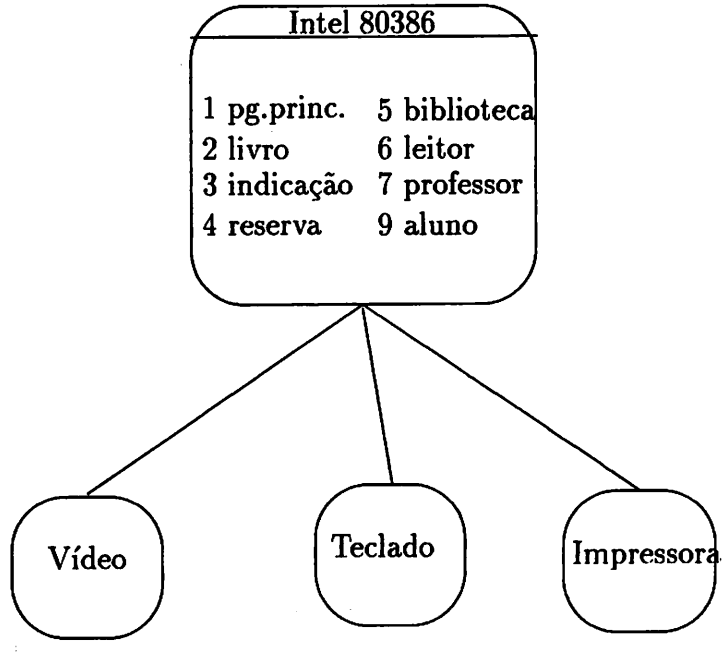


Figura 4.21: DP do Sistema de Biblioteca

Diagrama de Processos

Essa é a última etapa do projeto de um sistema como um todo, e corresponde também à última parte da arquitetura física do projeto – Arquitetura de Processos. Aqui são definidos quais os processos são executados nos processadores para o caso de haver mais de um. Supondo que o **Sistema de Biblioteca** seja desenvolvido para trabalhar em apenas um microcomputador (sistema monousuário), todos os processos deste sistema devem ser executados em apenas um processador, por exemplo, em um microprocessador da marca Intel 80386 para microcomputadores do tipo PC 386.

O Diagrama de Processos apresenta visualmente a disposição dos processos nos processadores e a disposição dos mecanismos utilizados na arquitetura do sistema. Em pequenos sistemas, como é esse estudo de caso, não existe necessidade de representação dessa arquitetura através do Diagrama de Processos (DP). Entretanto, para completeza da solução até aqui apresentada, a Figura 4.21 mostra esse Diagrama.

Capítulo 5

JSD-OO Tool: Uma Ferramenta de Apoio ao JSD-OO

O desenvolvimento de novas técnicas e o aperfeiçoamento de antigos métodos se fazem necessários à medida que o homem por sua própria natureza constrói sistemas cada vez mais complexos e portanto cada vez mais difíceis de serem gerenciados, mesmo quando utilizando novos métodos para construção desses sistemas.

O método JSD-OO sendo uma extensão natural do JSD, segue os mesmos passos para construção do modelo da realidade, utilizando entretanto as notações diagramáticas inerentes ao novo método. A construção de grandes sistemas que venham a utilizar o método JSD-OO, como em outras técnicas, deverá exigir muitos refinamentos e edição por parte do projetista, tornado-se uma tarefa dispendiosa e cansativa. Tendo isto em mente, o desenvolvimento de uma ferramenta CASE (Computer Aided Software Engineering) para o método JSD-OO, se torna de suma importância para auxiliar engenheiros de software que venham a se utilizar do novo método. Observa-se, por exemplo, no OOD o uso de vários ícones difíceis de serem feitos à mão para a representação dos conceitos e formalismos do paradigma de orientação a objetos. A ausência de uma ferramenta automatizada na construção de sistemas utilizando esse paradigma, diminui o nível de produtividade, além de aumentar a possibilidade de erros e piorar a confiabilidade geral do sistema gerado.

5.1 Aspectos Gerais da Ferramenta

Muitos dos requisitos necessários para as ferramentas CASE são independentes do método utilizado para o desenvolvimento de sistemas. Assim é que, por exemplo, todas as ferramentas modernas desse tipo devem fornecer facilidades gráficas para a construção da representação diagramática dos métodos de projeto.

O diagrama estrutural da Figura 5.1 dá uma visão geral dos componentes essenciais que uma ferramenta CASE para JSD-OO (e outros métodos) deve possuir [Fi90][1]. Os

[1]Maiores detalhes quanto às ferramentas CASE para projetos orientados a objetos podem ser obtidos em [Co91].

quatro componentes integrados teriam funções bem claras dentro da ferramenta, e são:

- suporte à *notação gráfica* do método JSD-OO, permitindo a representação de todos os símbolos que formam a visão diagramática do método;
- criação e manutenção de um *dicionário de dados* permitindo o armazenamento dos elementos que são a base do método, e fazendo uma verificação quanto a inconsistências que podem existir em relação à representação gráfica do modelo;
- um *gerador de telas*, que permita ao projetista a criação das telas de entradas e dos relatórios de saída, possibilitanto uma prototipagem mais eficiente do sistema;
- um *gerador de códigos*, que dê a capacidade a uma ferramenta CASE de gerar código compilável ou executável diretamente do projeto do sistema.

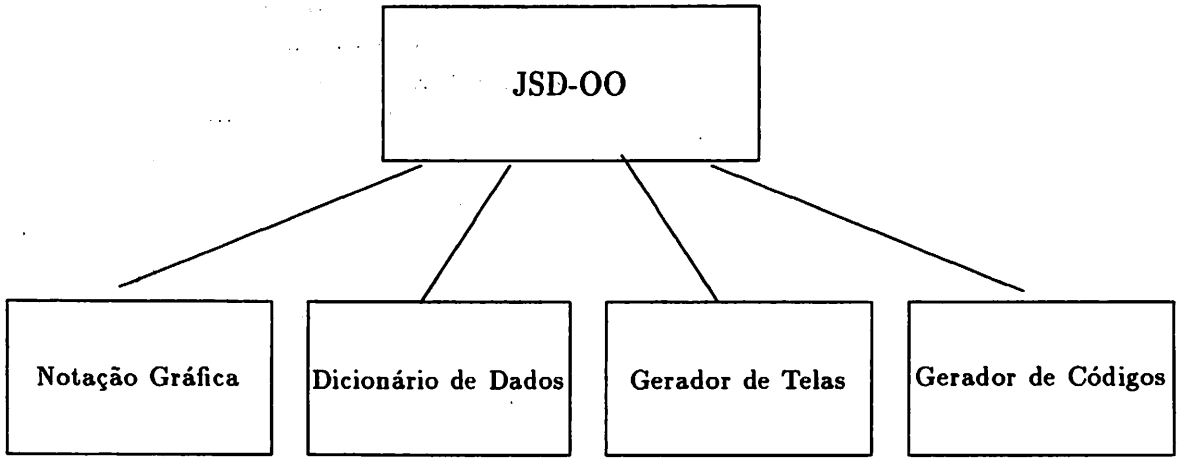


Figura 5.1: Principais Componentes da Ferramenta CASE para o Método JSD-OO

5.2 O JSD-OO Tool

Para mostrar as reais possibilidades de utilização da extensão do método JSD em projetos de sistemas, e sua adequação em modelar realidades passíveis de automatização, esse trabalho tomou como um de seus objetivos a especificação da estrutura de uma ferramenta de auxílio ao desenvolvimento de sistemas e a construção do protótipo dessa

ferramenta de auxílio para a elaboração de modelos JSD-OO, baseando-se no uso de todo o formalismo, visual e conceitual, do método.

O **JSD-OO Tool** como foi chamada essa ferramenta, foi desenvolvido para ser executado sob o ambiente gráfico Windows. Segundo pesquisa realizada pela revista PC Magazine, ele é o ambiente operacional com interface gráfica mais utilizado (mais de nove milhões de cópias vendidas) para plataformas de trabalho em máquinas padrão PC 286 ou superiores (PC Magazine Brasil 2(7):42-80 de 1992). Possui entre outras características, uma interface com o usuário baseada em ícones, tornando o uso desse sistema rápido nos controles e de aprendizado fácil e intuitivo.

A linguagem de programação escolhida para a implementação do protótipo foi uma linguagem orientada por eventos que produzisse uma interface padrão para ambiente Windows. Própria para desenvolvimento de aplicativos que trabalhem sob Windows, o Visual Basic possibilita a criação, de modo rápido e produtivo, de interfaces do usuário. O Visual Basic faz parte do grupo de linguagens responsáveis por uma nova fase no desenvolvimento de aplicativos, a era da programação visual. O surgimento das interfaces gráficas deu origem à novas ferramentas de programação, com ênfase na abordagem visual e utilização intensa por parte dos usuários de periféricos, como por exemplo o mouse, para a escolha das tarefas.

5.2.1 Uma Aplicação

O exemplo que se segue foi escolhido para demonstrar o uso da ferramenta JSD-OO Tool e é o mesmo do estudo de caso do método JSD-OO, entretanto para efeito de simplficação, não iremos apresentar todas as telas necessárias para desenvolvimento do modelo através da ferramenta, detalhando apenas as mais importantes para fins de conhecimento do padrão de trabalho da ferramenta.

As Figuras 5.2 e 5.3 mostram as telas de apresentação e inicial com a barra de opções do Menu Principal suspensa no topo da tela.

A Figura 5.4 detalha as propriedades da entidade **Reserva** introduzida no Dicionário de Dados da ferramenta, mostrando os atributos e as ações requeridas e sofridas. Observa-se nesta janela a possibilidade de as entidades com ascendente direto poderem herdar os atributos e as ações sofridas da entidade pai. Na figura, **Reserva** não possui ascendente ficando o campo **Entidade Pai:** vazio.

A Figura 5.5 mostra o Diagrama de Estruturação Cronológica da entidade **Reserva** pronto, dentro do espaço definido pela ferramenta para desenho dos diagramas. Abaixo de cada diagrama observa-se os botões para inclusão dos elementos de controle **Seleção**, **Sequência**, e **Repetição** e mais o botão **Excluir** para exclusão de elementos introduzidos incorretamente.

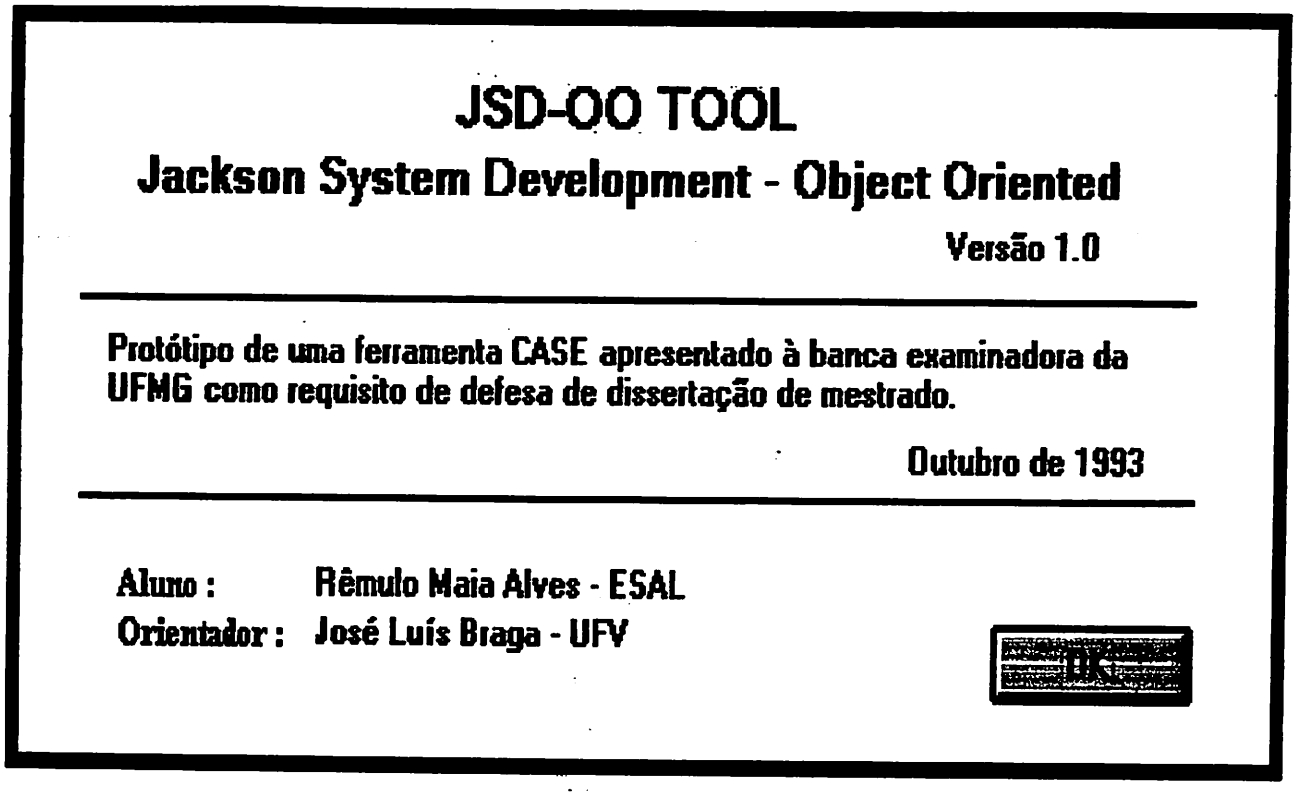


Figura 5.2: Tela de Apresentação do JSD-OO Tool

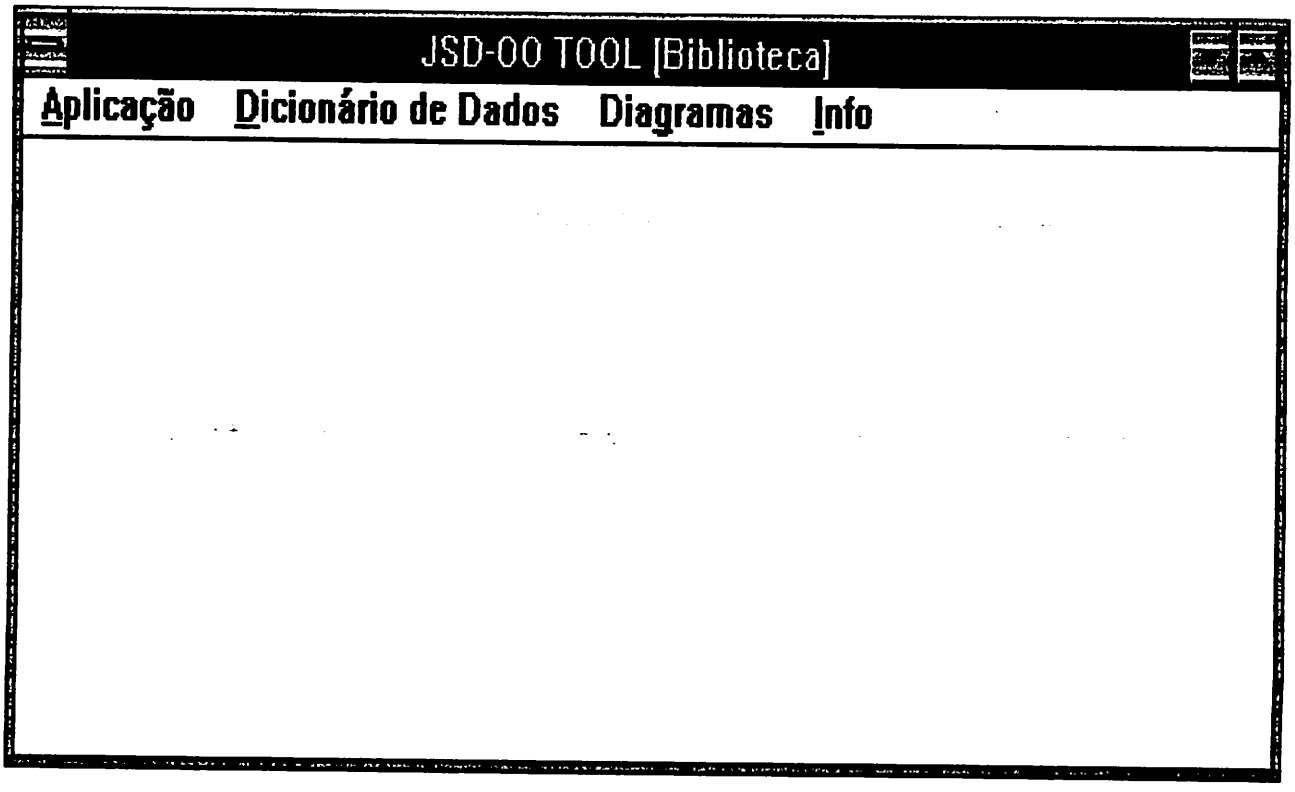


Figura 5.3: Tela Principal do JSD-OO Tool

A seguir, a Figura 5.6 mostra o Diagrama de Herança do modelo estudado, com os tipos-entidade **Professor** e **Aluno** apontando para seu ancestral comum o tipo-entidade **Leitor**. Como pode ser observado, a tela não mostra qualquer comando de botão que possibilite a edição do diagrama por parte do usuário pois sua construção é totalmente automática, com base no dicionário de dados.

A Figura 5.7 apresenta o Diagrama de Especificação de Sistemas do modelo inicial do estudo de caso, através do uso do JSD-OO Tool. Verifica-se na tela, na parte inferior da janela, a disposição da barra de comandos mostrando os botões **Seta**, **Orientação**, **Duplicar** e **Excluir** usados na montagem do DES. **Seta** introduz no diagrama o ícone *seta* com o objetivo de orientar as ações executadas ou sofridas do modelo. O botão **Orientação** coloca o sentido de direção do ícone introduzido anteriormente. Os botões **Duplicar** e **Excluir** servem na edição das entidades e do nome das ações do modelo.

A Figura 5.8 apresenta o Diagrama de Módulos para a modelagem feita sobre o caso da biblioteca do estudo de caso. Os botões de comando **Corpo Principal**, **Corpo Módulo**, **Corpo Interface**, **Entidades**, **Associar...** e **Excluir** posicionados na parte inferior da tela estão associados ao controle e edição dos ícones para construção do Diagrama de Módulos. O primeiro botão diz respeito ao posicionamento do programa principal (Corpo Principal) em relação aos módulos restantes; os outros dois botões (Corpo Módulo e Corpo Interface) posicionam as entidades do sistema em relação aos módulos juntamente com a implementação ou apenas com a declaração da sua interface. Os botões **Entidades** e **Associar...** estão associados à apresentação e especificação das entidades distribuídas nos módulos e o botão **Excluir** elimina módulos mal projetados do diagrama.

A Figura 5.9 apresenta o Diagrama de Processos do estudo de caso, através do JSD-OO Tool. Os botões de comando **Processador**, **Dispositivo**, **Módulos**, **Associar...** e **Excluir** localizados na parte inferior da tela, possibilita o gerenciamento do diagrama. Os botões **Processador** e **Dispositivo** permitem introduzir, respectivamente, os símbolos para processador e dispositivo de entrada e saída de dados no formulário de trabalho do Diagrama de Processos. O botão **Módulos** apresenta os módulos associados a cada processador do sistema; o botão **Associar...** edita os módulos associados a um processador específico e o botão **Excluir** elimina um símbolo qualquer do formulário do diagrama.

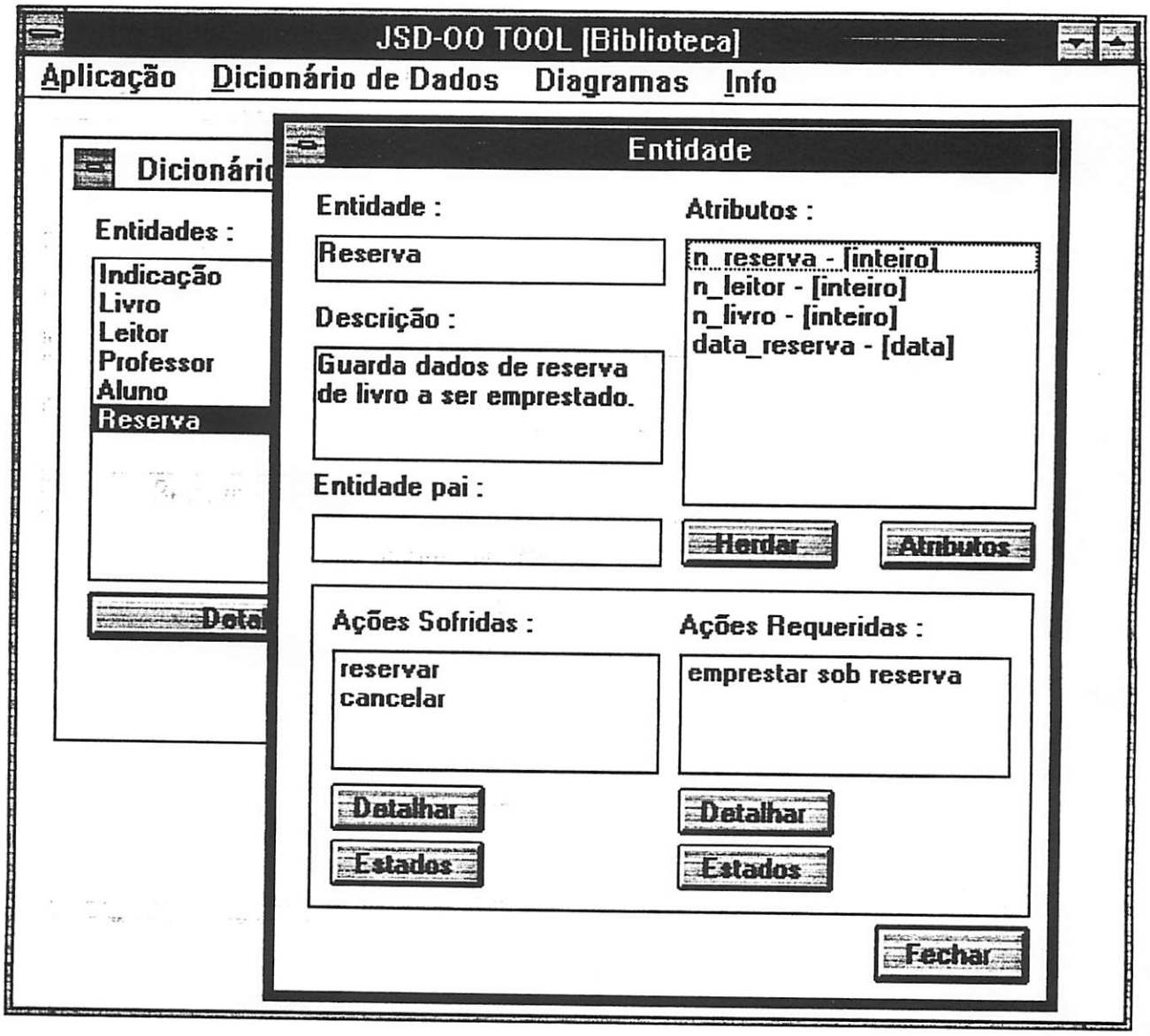


Figura 5.4: Tela de entrada da Entidade Reserva no DD

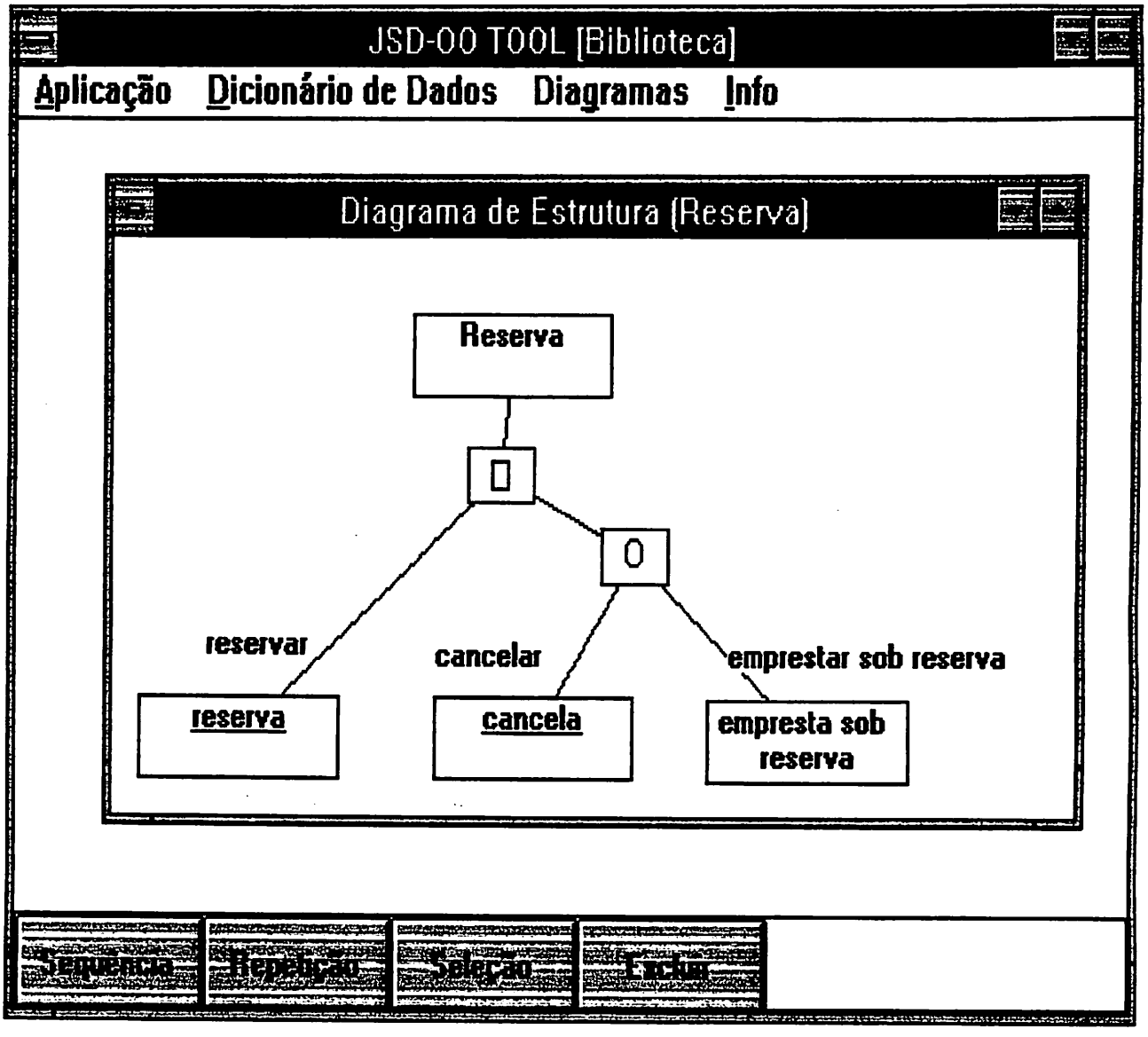


Figura 5.5: Tela com o DEC da Entidade Reserva

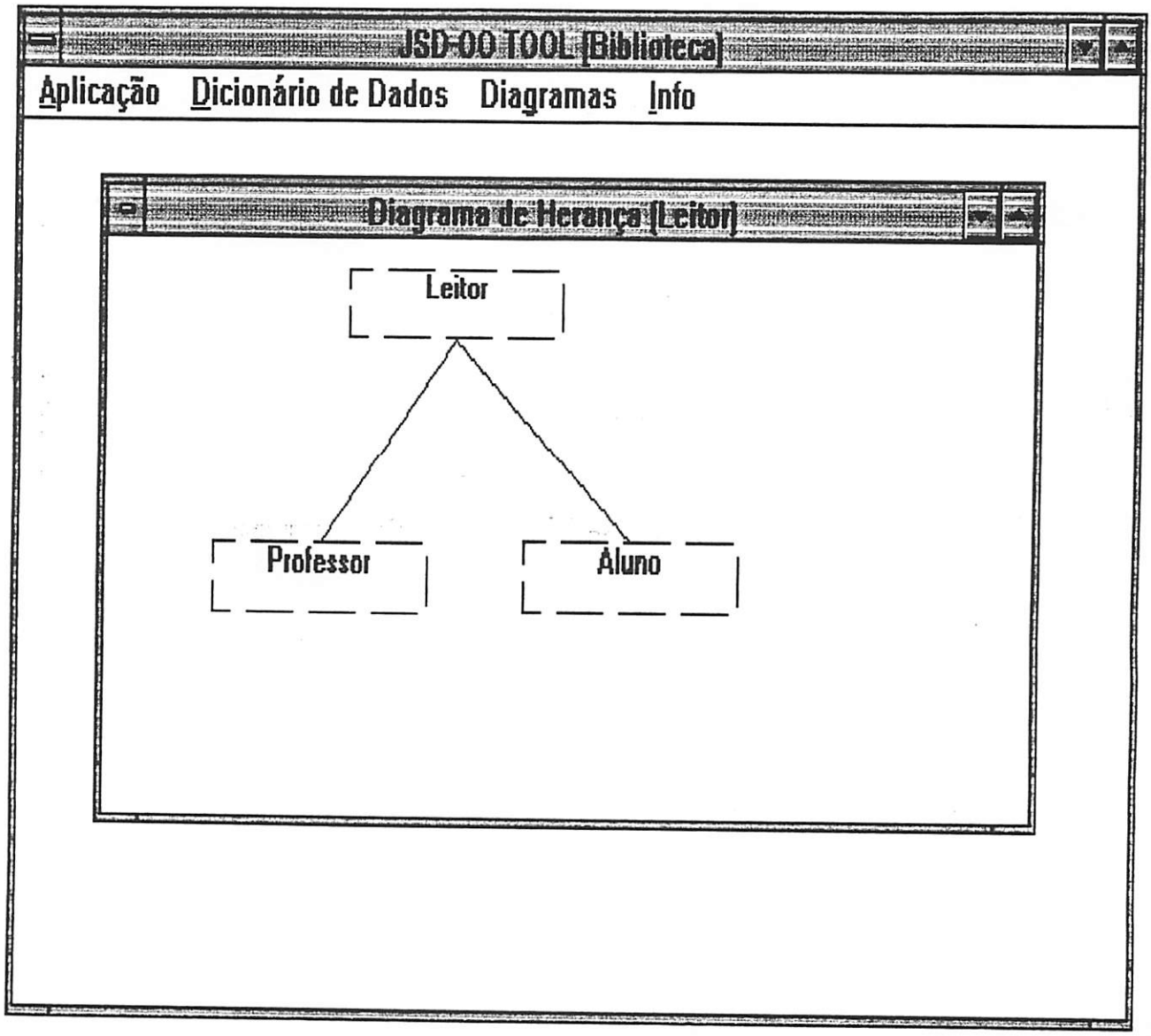


Figura 5.6: Tela com o Diagrama de Herança

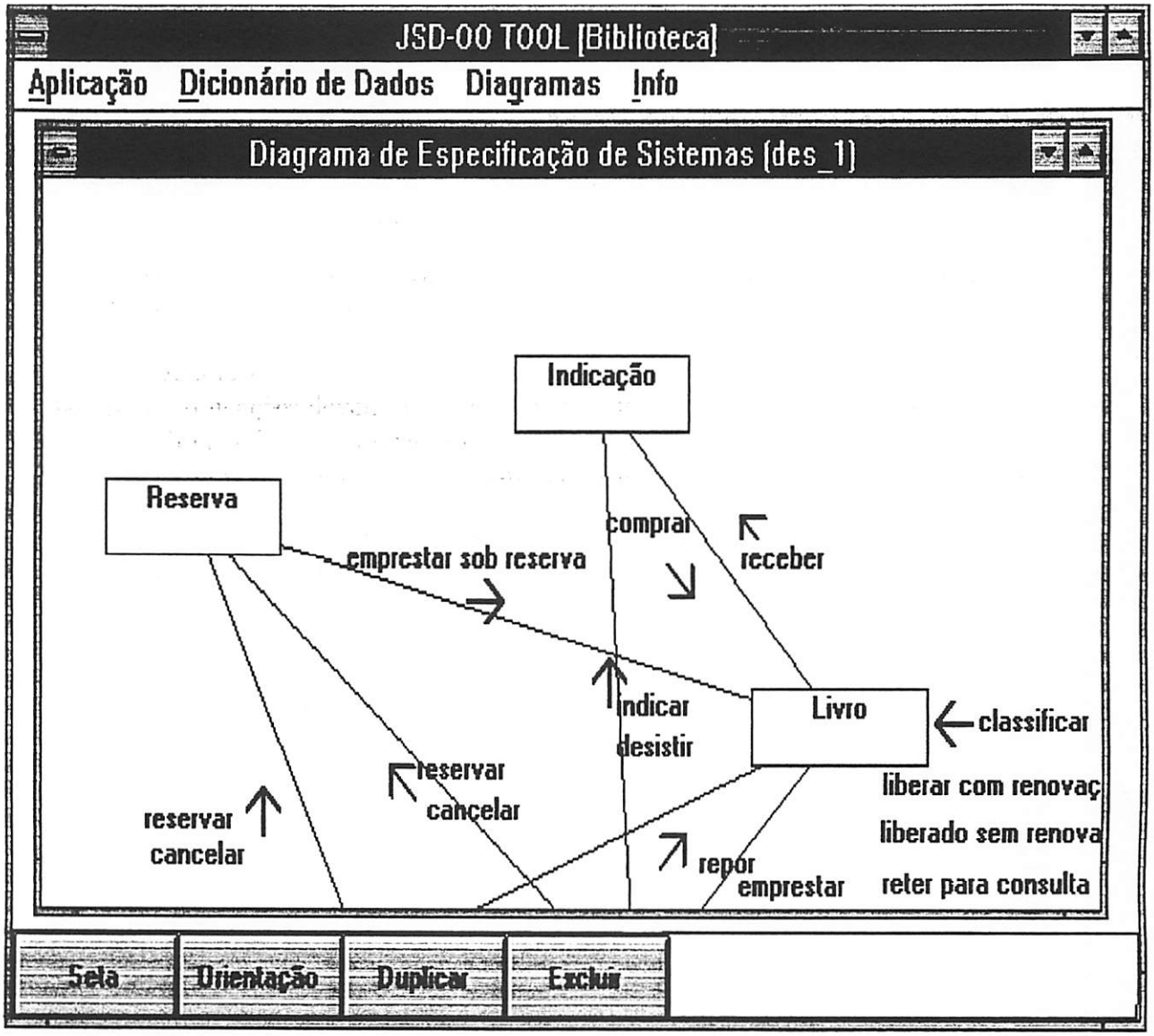


Figura 5.7: Tela mostrando parte do DES do Modelo Inicial

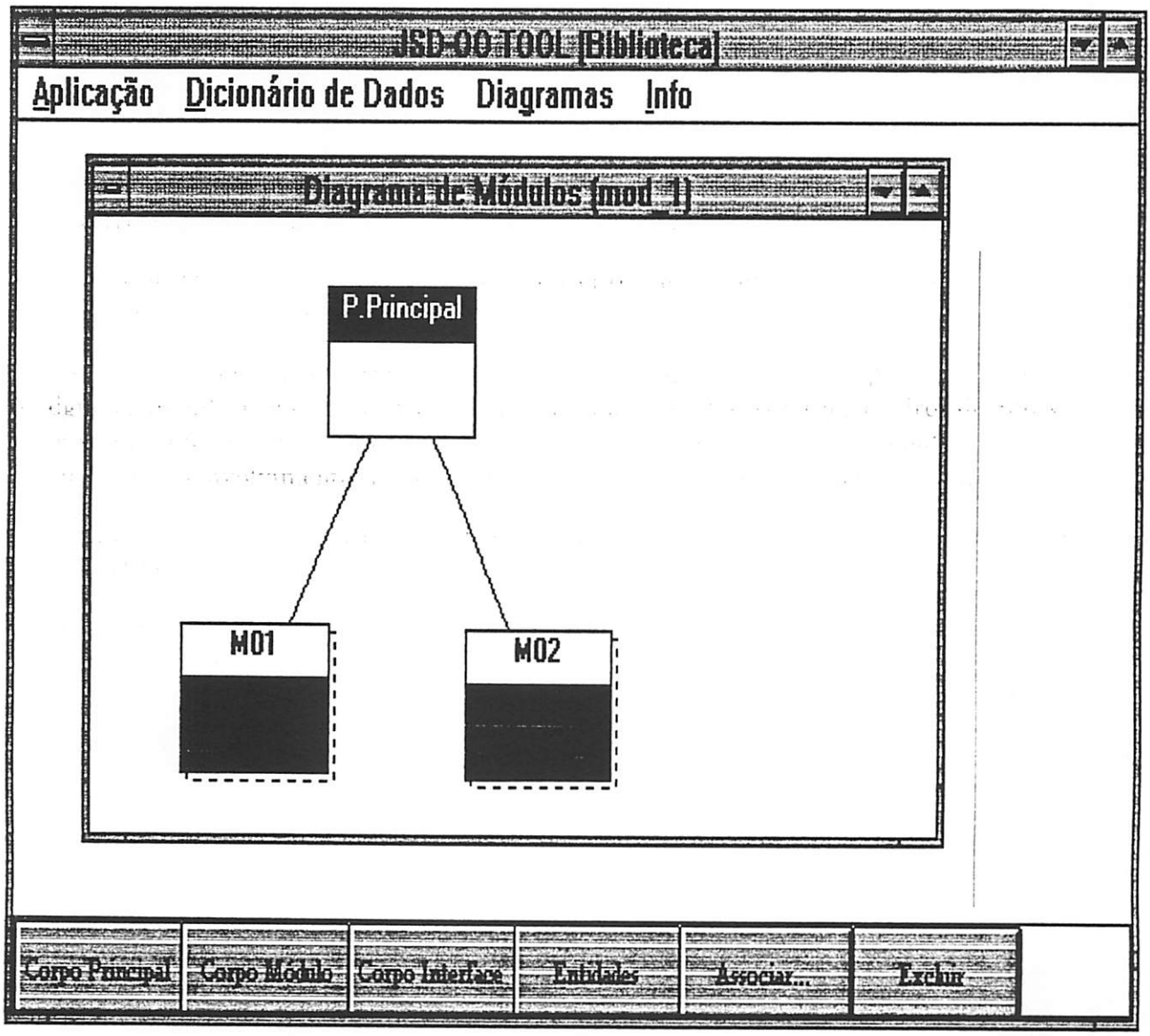


Figura 5.8: Tela com o Diagrama de Módulos

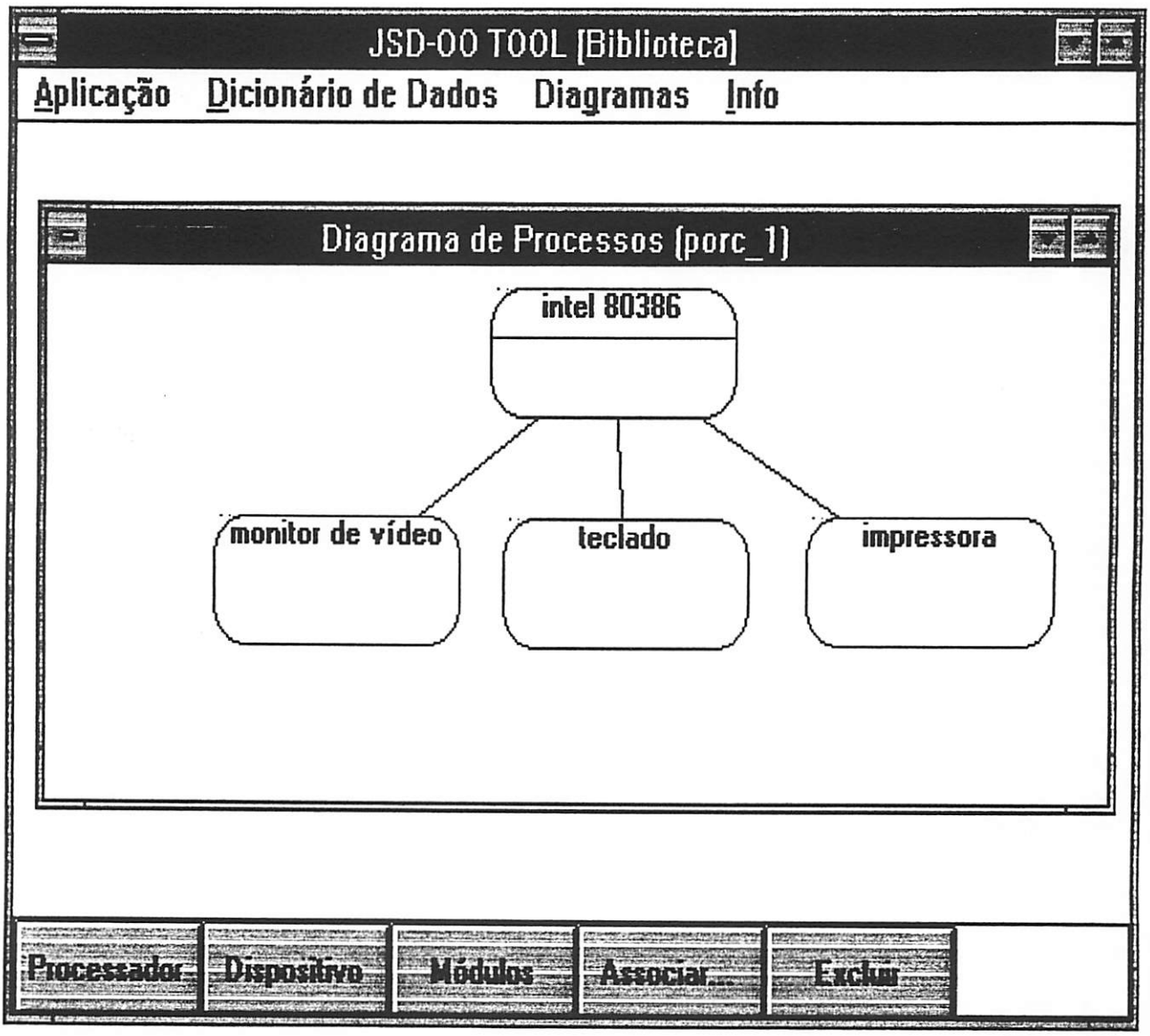


Figura 5.9: Tela com o Diagrama de Processos

Capítulo 6

Conclusões

6.1 Avaliação do Trabalho

O método JSD-OO pretende ser uma extensão do método JSD, que permita sua utilização plena como método Orientado a Objetos. Sua simplicidade e facilidade de uso, aliados ao pequeno número de conceitos a serem aprendidos para que seja bem utilizado, o tornam candidato para uso em modelagem de sistemas.

Não foi objetivo do trabalho a criação de um novo método, totalmente diferente de todos os existentes e em uso, mas tão somente a definição, através de comparações, de um conjunto de extensões ao já conhecido JSD. Essas extensões surgiram da análise das características dos métodos orientados a objetos existentes e em uso, e da comparação dessas características com as semelhantes no JSD.

As principais extensões ao JSD, aqui apresentadas e discutidas, foram:

- inclusão do diagrama de herança como parte da fase II do método JSD;
- eliminação das fases V e VI do método JSD com a inclusão em seu lugar de conceitos e técnicas oriundos da visão física que o modelo de objetos oferece através dos diagramas de módulos e de processos do método OOD;
- adptação do DEC do método JSD para mostrar os possíveis estados de cada entidade.

A expectativa é a de que o JSD-OO possa ser bastante utilizado como método de análise e projeto de sistemas baseado em objetos, principalmente devido à sua simplicidade, economia de conceitos, notação simples e concisa, que são requisitos mínimos para que um método se firme entre seus usuários. Numa comparação entre os métodos de projeto OOD e JSD-OO em relação às características acima descritas podemos observar uma clara vantagem para a utilização plena do método JSD-OO considerando-se o aprendizado de novos usuários.

6.2 Extensões ao Trabalho

Em seqüência a este trabalho, outros pontos serão analisados, e novos recursos adicionados ao JSD, tornando-o mais aderente ao paradigma de orientação a objetos. Podem ser citados os seguintes aspectos adicionais:

- um estudo mais aprofundado da fase de implementação do método JSD com vistas à possibilidade do aproveitamento do processo de desenvolvimento do Diagrama de Implementação de Sistemas (DIS);
- possível substituição do texto estruturado por uma ferramenta que produza uma documentação mais consistente com os conceitos da orientação a objetos, como as estruturas para documentação do método OOD por exemplo;
- possibilidade de se introduzir novos conceitos não abordados neste trabalho como polimorfismo, persistência, etc.

Em relação à ferramenta JSD-OO Tool e em conformidade com o exposto no Capítulo 5, algumas modificações devem ser efetivas para que Projetistas e Engenheiros de Softwares possam obter máximo aproveitamento do método e portanto um aumento de produtividade no desenvolvimento de novos sistemas. Entre essas modificações destacamos:

- Modelar sistemas a partir da utilização dos diagramas de representação do método JSD-OO;
- Geração automática de códigos executáveis ou compiláveis que implementem o modelo JSD-OO;
- Geração das telas de entrada e dos relatórios de saída

Apêndice A

Texto Estruturado

Texto Estruturado é uma forma textual, alternativa, utilizada para representar os Diagramas de Estruturação Cronológicas (DEC) do método JSD. M.A. Jackson trouxe essa forma de representação do método para projeto de programas JSP [Ja75], alterando apenas o seu nome original de *Lógica Esquemática* para *Texto Estruturado.*

Ela é comparada à forma usada no *Português Estruturado* para representação de processos lógicos nos métodos tradicionais de desenvolvimento estruturados, diferindo das linguagens de programação convencionais (Cobol, Pascal, Fortran) por limitar o uso de comandos de desvios lógicos à um pequeno grupo de estruturas de controle. Com isso essa forma permite:

1. maior grau de independência do projeto em relação à linguagem de programação utilizada;

2. maior grau de documentação do projeto, facilitando em muito a fase de manutenção do ciclo de vida do sistema.

O Texto Estruturado pode ser realizado em paralelo à representação diagramática do sistema em desenvolvimento, servindo como instrumento de auxílio para o responsável pelo projeto na apresentação da especificação funcional aos usuários do sistema, pois na maioria das vezes essa forma de representação se torna mais legível para o entendimento.

As estruturas usadas para representar controle no método JSD são definidos como abaixo:

- Sequência: as partes a serem executadas em uma sequência são postas em ordem de execução do início ao fim do texto do processo. A forma gramatical da sequência é:

  X seq
  ...
  X end

Exemplo:

Matrícula seq
    Pagar Taxa;
    Apanhar Formulário de Inscrição;
    Preencher Formulário;
    Obter Assinatura do Coordenador;
    Devolver Formulário à Secretaria do Curso;
Matrícula end

- Seleção: as condições da estrutura são estabelecidas nas expressões sel e alt. A forma gramatical é:

X sel
    ...
X alt
    ...
X end

Exemplo:

Depósito sel (registro de créditos)
    Adicionar Valor ao Saldo;
Depósito alt (registro de débito)
    Subtrair Valor do Saldo;
Depósito end

- Repetição: a condição para o teste é estabelecida na expressão itr, e a estrutura pode ser repetida zero ou mais vezes. Uma observação em relação a essa e as estruturas de controle anteriores, é que nas formas gramaticais do texto estruturado as ações ou componentes elementares nos diagramas, são terminados com um ponto e vírgula. A forma gramatical para repetição é:

X itr
    ...
X end

Exemplo:

X itr (enquanto houver registros de créditos a serem processados)
    Adicionar Valor ao Saldo;
X end

Apêndice B

Esquemas do Método OOD

Nesse apêndice são mostrados as formas alternativas de representação utilizados pelo Método OOD em complemento à representação gráfica utilizada para representar as várias visões do modelo da realidade.

B.1 Estruturas de Classes

Esquema para Definição de Classe :

```
Nome                    : (identificador)
Documentacao            : (texto)
Visibilidade            : (exportado / privativo / importado)
Cardinalidade           : (0 / 1 / n)
Hierarquia              :
  Superclasses:(lista de nomes de classes)
  Metaclasse  :(nome da classe)
Parametros Genericos:(lista de parametros)
Interface / Implementacao
(Publico / Protegido / Privativo):
      Usa       :(lista de nome de classes)
      Campos    :(lista de declaracoes de campos)
      Operacoes:(lista de declaracoes de operacoes)
Maquina de Estado Finito:(diagrama de transicao de estado)
Concorrencia:(sequencial / bloqueada / ativa )
Complexidade de espaco:(texto)
Persistencia:(persistente / transitoria)
```

Esse Esquema captura todos os aspectos importantes da descrição de uma classe. Na prática o projetista não precisa preencher todo o Esquema, a menos que todos os itens sejam importantes.

A visibilidade (terceiro elemento), indica se a classe é exportada, privativa, ou importada em relação a categoria de classe a que ela pertence.

A cardinalidade (quarto elemento) da classe captura quantas instâncias dessa classe serão permitidas.

O papel que essa classe desempenha na hierarquia é expresso pelos dois próximos elementos do Esquema. Uma classe pode ter zero, uma ou mais Superclasses e zero ou uma Metaclasse.

O próximo elemento fornece os parâmetros de passagem da classe.

Os três campos – *Usa*, *Campos* e *Operações* – podem ser repetidos quatro vezes: três vezes para a interface e uma vez para a implementação da classe. A interface de uma Classe (se a linguagem permitir) pode ser dividida em três partes : *pública*, *privada*, e *protegida*. Esse é o elemento mais importante de um Esquema de classe, pois é aqui que capturamos a visão externa da classe, isto é, todas as propriedades (os elementos declarados em *Campos* no Esquema), bem como todas as operações da classe.

O elemento *Operações* contém uma lista de operações definidas na classe, e que também possuem o seu próprio Esquema.

O elemento *Usa* é similar aos elementos *Superclasses* e *Metaclasse* .

Nesse Esquema os quatro próximos elementos representam a semântica dinâmica e comportamento de tempo e espaço da classe. O elemento *máquina de estado finito* representa o Diagrama de Transição de Estado.

O elemento *Concorrência* documenta se as instâncias da classe são sequenciais, bloqueadas ou ativas. Evidentemente esta semântica deve ser coerente com a semântica associada a cada operação.

O elemento *complexidade de espaço* documenta o espaço consumido pelas instâncias da classe e pode ser expresso em unidades de memória ou em termos relativos (notação O(n)).

O último elemento documenta a *persistência* de objetos dessa classe. Um *objeto persistente* é aquele cujo estado e classe persistem além do tempo de vida do programa que o criou, um *objeto transitório* é aquele cujo tempo de vida não vai além do programa que o criou.

Esquema para Definição de Classe de Utilitários :

```
Nome          : (identificador)
Documentacao  : (texto)
Visibilidade  : (exportado / privativo / importado)
Parametros Genericos : (lista de parametros)
Interface/Implementacao :
        Usa       : (lista de nome de classes)
```

```
Campos     : (lista de declaracoes de campo)
Operacoes  : (lista de declaracoes de operacoes)
```

Desde que a classe de utilitários representa uma coleção de subprogramas livres (não definidos dentro de uma classe específica) o Esquema para classe de utilitários é um pequeno subconjunto do Esquema para classes. Existe apenas uma parte *interface* e uma *implementação.*

Esquema para Definição de Operações :

```
Nome            : (identificador)
Documentacao    : (texto)
Categoria       : (texto)
Parametros Formais:lista de declaracoes de parametros
Resultado     :(nome da classe)
Precondicoes:(PDL/diagrama Objeto)
Acao          :(PDL/diagrama Objeto)
Poscondicoes:(PDL/diagrama Objeto)
Excecoes      :(lista de declaracoes de excecoes)
Concorrencia:(sequencial /protegida /concorrente /multipla)
Complexidade de tempo :(texto)
Complexidade de espaco:(texto)
```

Nem sempre há necessidade de um Esquema de operações para cada operação declarada no Esquema de classes ou de classe de utilitários, bastando apenas o nome das operações, seus parâmetros, e o seu significado em formato livre de texto.

Os primeiros dois elementos são auto-explicativos. O terceiro elemento, *categoria*, fornece o modo como operações relacionadas são agrupadas logicamente. Por exemplo operações que modificam o estado de um objeto podem ser agrupadas na categoria denominada como *Modificadora* (ver página 16).

Completamos a visão estática das operações listando seus parâmetros formais e resultado (apenas para funções).

A semântica dinâmica de cada operação é documentada nos quatro elementos seguintes do Esquema para operações. Devemos documentar o significado de cada operação ou de uma forma informal, usando texto livre no elemento *Ação*, ou formalmente pelos elementos *Précondições*, *Póscondições* e *Exceções.*

A semântica de uma operação pode às vezes apontar para outro diagrama de objeto que mostra o relacionamento entre os objetos participantes na operação e a semântica dinâmica da operação, expressos através dos Diagramas de Temporização ou de uma linguagem de projeto de programa (PDL) (página 28).

Os últimos três elementos do Esquema são *concorrência semântica*, *complexidade de tempo* e *complexidade de espaço* de cada operação. Uma operação pode ser sequencial, significando que sua semântica é garantida somente em presença de uma simples linha de controle. Alternativamente, a semântica de uma operação pode ser garantida quando chamada por múltiplos processos, mas com tratamentos diferentes para o problema de sincronização: *protegida*, *concorrente* ou *múltipla*.

B.2 Esquema para Definição de Diagrama de Transição de Estados

```
Eventos       : (lista de identificadores)
Documentacao  : (texto)
Acao          : (PDL/Diagrama Objeto)
```

B.3 Estruturas de Objetos

Esquema para Definição de Objeto :

```
Nome          : (identificador)
Documentacao  : (texto)
Classe        : (nome da classe)
Persistencia  : (persistente / estatico / dinamico)
```

O Esquema para definição de objeto visto acima documenta a classe à qual o objeto pertence, informando entre outras coisas o tipo de persistência do objeto, que deve ser coerente com a qualidade de persistência definida na classe. Ex.: se o Esquema para definição de uma classe estabelecer que todas as instâncias da classe são *transitórias* então a persistência dos objetos dessa classe só podem ser do tipo *estático*, isto é, os objetos só existem durante a execução do programa, ou então do tipo *dinâmico*, isto é, os objetos são criados e destruídos dinâmicamente durante a execução do programa. Se por outro lado o Esquema para definição de uma classe indicar que os objetos dessa classe são *persistentes* então os objetos podem ser estáticos, dinâmicos, ou persistentes (os objetos continuam a existir mesmo depois que o programa que os criou acabar).

Esquema para Definição de Mensagem :

```
Operacao      :(nome da operacao)
Documentacao  :(texto)
```

```
Frequencia    :(aperiodica / periodica)
Sincronizacao:(simples/sincronizada/frustrada/
               limitada/assincrona)
```

O Esquema para definição de mensagem, visto acima, especifica a operação definida a partir da classe do objeto, que por sua vez fornece detalhes semânticos da operação. Outro ponto mostrado no Esquema é a frequência com que a mensagem é enviada.

B.4 Arquitetura de Módulo

Esquema para Definição de Diagrama de Módulo :

```
Nome          : (identificacao)
Documentacao  : (texto)
Declaracoes   : (lista de declaracoes)
```

A mais importante documentação associada com um módulo é uma lista das várias declarações que ele contêm, que podem incluir classes, classes de utilitários, objetos, etc. Devemos declarar apenas as classes e objetos mais importantes para os módulos.

B.5 Arquitetura de Processo

Esquema para Definição de Processador :

```
Nome            : (identificador)
Documentacao    : (texto)
Caracteristicas : (texto)
Processos       : (lista de processos)
Escalonamento   : (preemptivo / nao preemptivo / ciclico
                   / executivo / manual)
```

Para cada processador devemos documentar elementos como por exemplo, as características do computador (fabricante, no. do modelo, quantidade de memória, etc). Podemos documentar os processos escolhidos para serem alocados nesse processador,etc.

Esquema para Definição de Processo :

```
Nome          : (identificador)
Documentacao  : (texto)
Prioridade    : (inteiro)
```

Usamos um Esquema para definição de processo para documentar apenas uma linha de controle. Devemos incluir também sua prioridade relativa, se existir uma.

Uma implementação simples pode ter somente um processo, mas se o sistema é mais complicado pode haver dúzias de processos de uma só vez. Nem todos os processos podem ser ativos simultâneamente, mas utilizando o Esquema para definição de processador para coletar as informações sobre todos os processos em potencial podemos escalonar os processos da melhor maneira possível.

Esquema para Definição de Dispositivo :

```
Nome            : (identificador)
Documentacao    : (texto)
Caracteristicas : (texto)
```

Apêndice C

Estudo de Caso utilizando o Método JSD

C.1 Fase I - Escolha das Entidades e Ações

Esse problema está enunciado originalmente no livro "Análise Estruturada de Sistemas pelo Método de Jackson" do Professor Paulo C. Masiero [Ms92]. O texto é o mesmo do estudo de caso do método JSD-OO (Página 49).

Entidade: Indicação

Ações e Atributos das Ações:
Indicar: n_indic, título, autor, editora, n_edic, data_publ, data_indic, leitor_q_indic
Reindicar: n_indic, leitor_q_indic
Desistir: n_indic, leitor_desist
Encomendar: n_indic, data_encom, vendedor, preço
Receber: n_indic, data_receb
Atributos da Entidade: n_indic, título, autor, editora, n_edic, data_publ, data_indic, leitor_indic, data_aquis, preço

Entidade: Livro

Ações e Atributos das Ações:
Comprar: n_indic, data_aquis
Repor: livro_rep, data_rep
Doar: livro_doado, data_doac, título, autor, editora, n_edic, data_publ
Classificar: livro, classificação
Liberar com renovação: livro
Liberar sem renovação: livro
Emprestar: livro, leitor, data_empr
Renovar: livro, data_renov

Devolver: livro, data_devol
Reter para Consulta: livro, data_retenc, período_ret
Retirar: livro, leitor, data_retir
Retornar: livro
Perder: livro, data_perda
Expurgar: livro, data_expur, motivo_expur
Atributos da Entidade: n_livro, n_indic, data_aquis, preço, título, autor, editora, n_edic, data_publ, classificação, data_empr, data_devol, período_ret, data_perda, data_expur, motivo_expur

Entidade: Leitor

Ações e Atributos das Ações:
Emprestar Normal: (os mesmos atributos de emprestar de livro)
Emprestar Reserva: (os mesmos atributos de emprestar de livro) + reserva
Renovar: (os mesmos atributos de renovar de livro)
Devolver: (os mesmos atributos de devolver de livro)
Retirar: (os mesmos atributos de retirar de livro)
Retornar: (os mesmos atributos de retornar de livro)
Reservar: (os mesmos atributos de reservar de reserva)
Cancelar: (os mesmos atributos de cancelar de reserva)
Inscrever: n_insc, nome_leitor, categoria, matrícula, identidade, validade, data_insc
Atualizar dados: (os mesmos atributos de inscrever)
Desligar: n_insc
Atributos da Entidade: n_insc, nome_leitor, categoria, matrícula, identidade, ender, validade, data_insc

Entidade: Reserva

Ações e Atributos das Ações:
Emprestar reserva: (os mesmos atributos de emprestar reserva de leitor)
Reservar: n_reserva, n_leitor, livro, data_reserva
Cancelar: n_reserva, n_leitor
Atributos da Entidade: n_reserva, n_leitor, livro, data_reserva

C.2 Fase II - Estrutura das Entidades

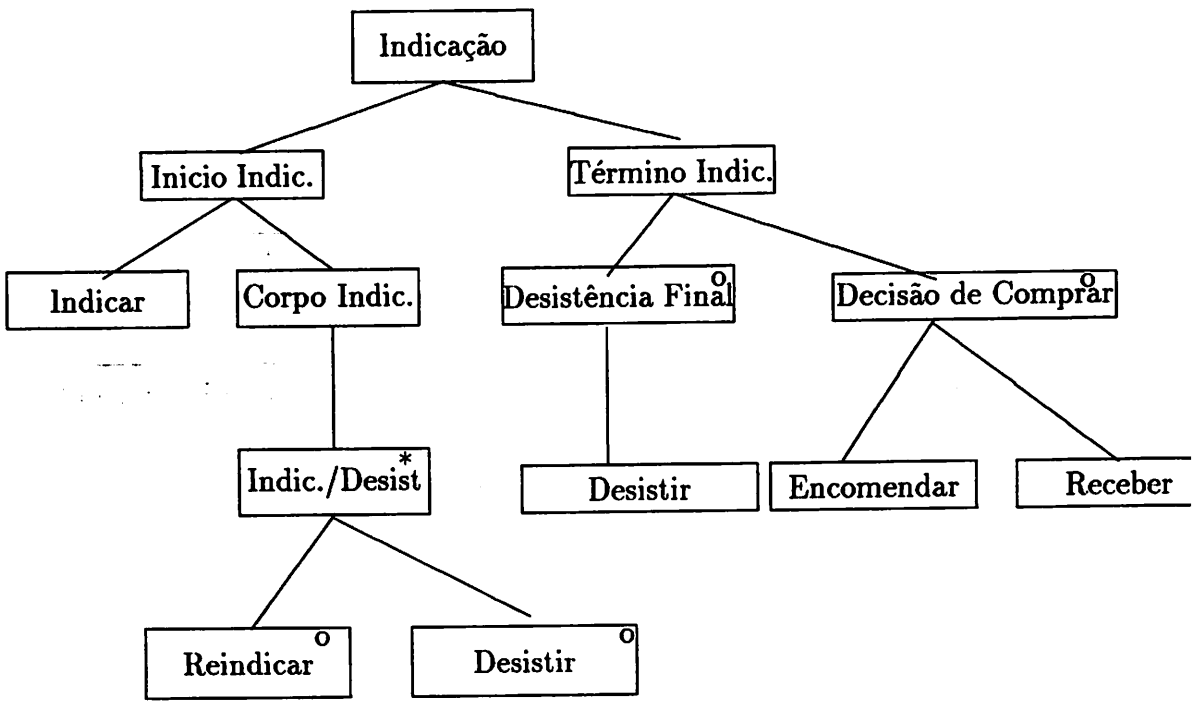


Figura C.1: DEC da Entidade Indicação

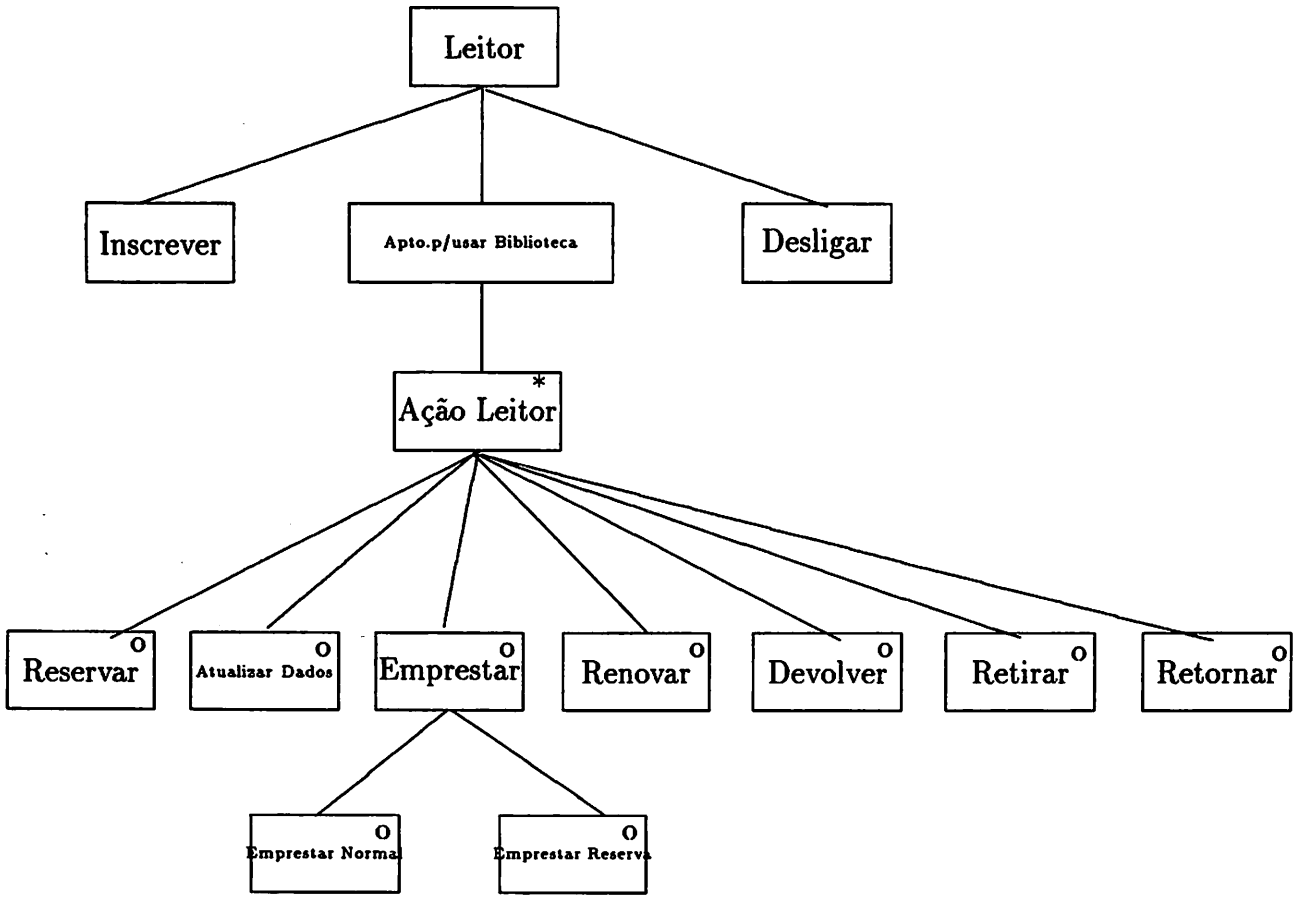


Figura C.2: DEC da Entidade Leitor

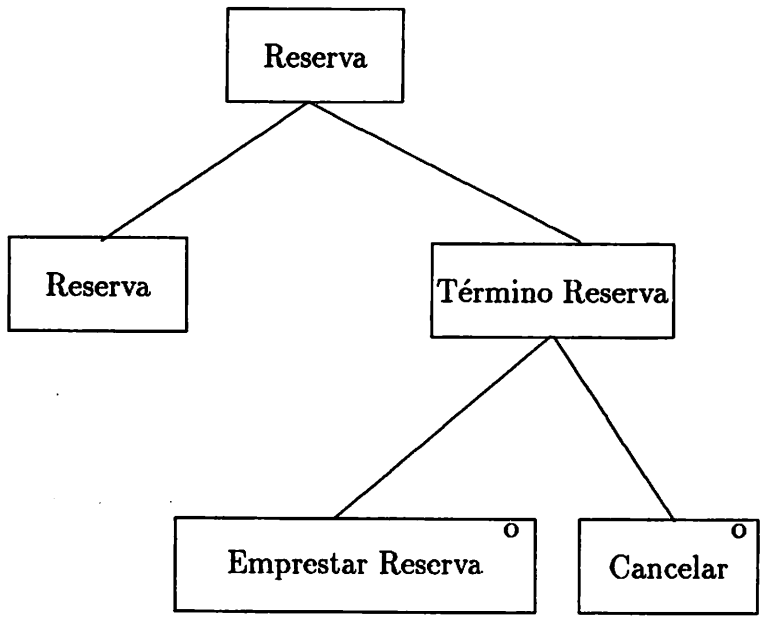


Figura C.3: DEC da Entidade Reserva

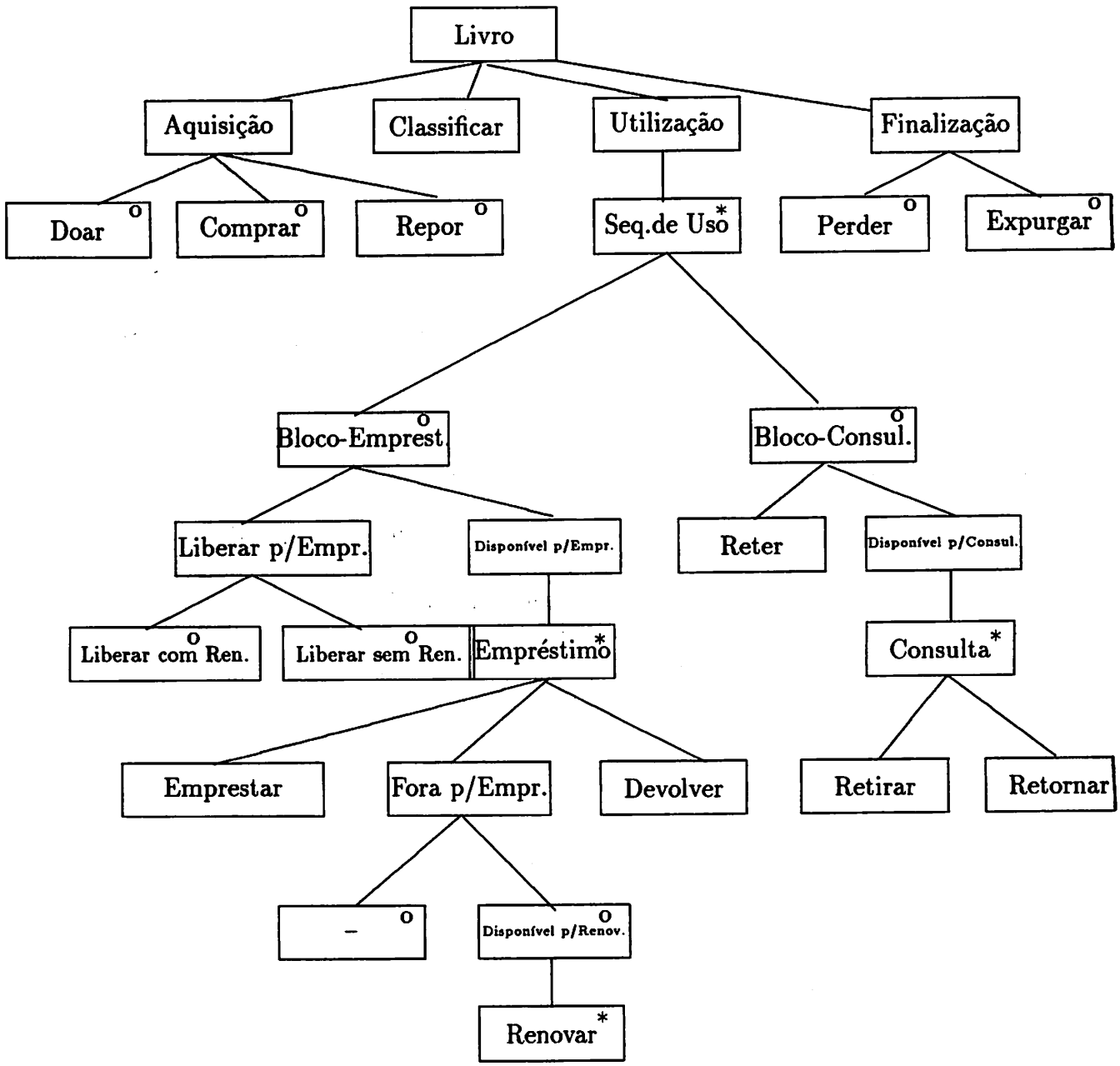
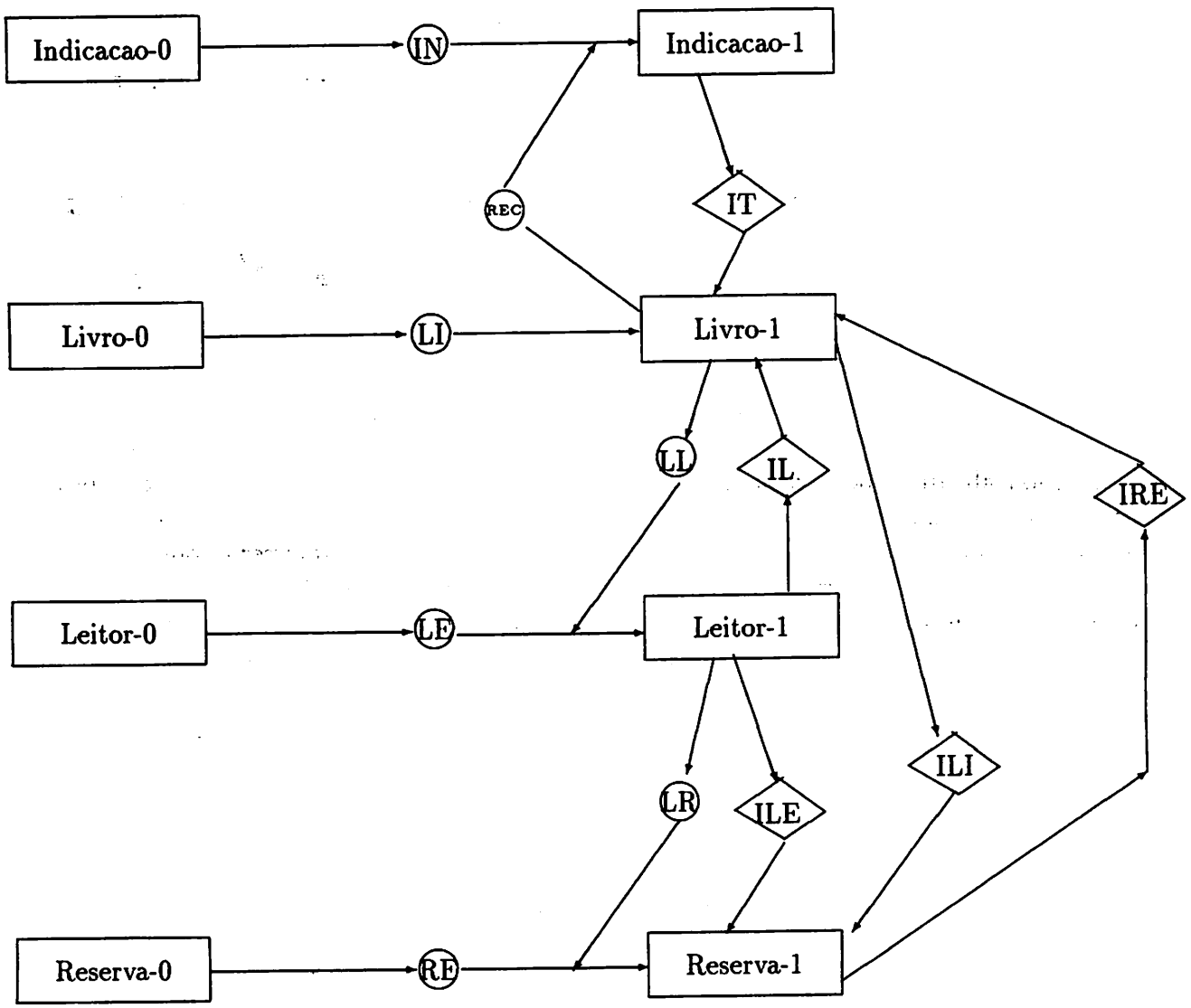


Figura C.4: DEC da Entidade Livro

C.3 Fase III - Construção do Modelo Inicial

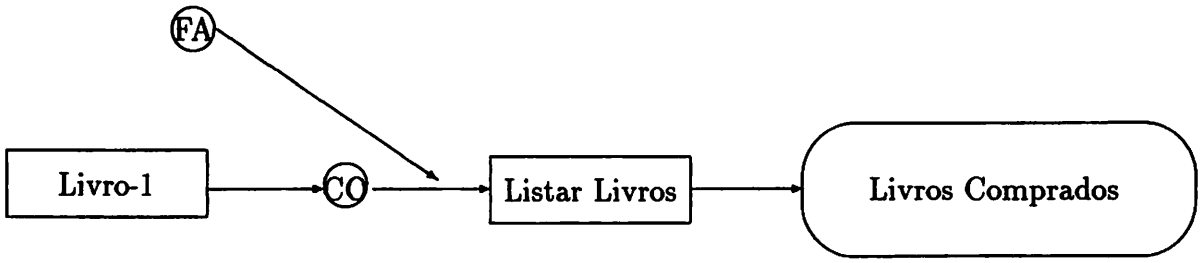


Os vetores de dados e sequências de dados usados no modelo inicial acima têm o seguinte significado:

IN ações de Indicação
LI ações de Livro
LE ações de Leitor
RE ações de Reserva
REC dados de Livro-1 para Indicação-1
LL dados de Livro-1 para Leitor-1
LR dados de Leitor-1 para Reserva-1
ILE informações sobre estado de Leitor-1
ILI informações sobre estado de Livro-1
IT informações sobre estado de Indicação-1
IL informações sobre estado de Leitor-1
IRE informações sobre estado de Reserva-1

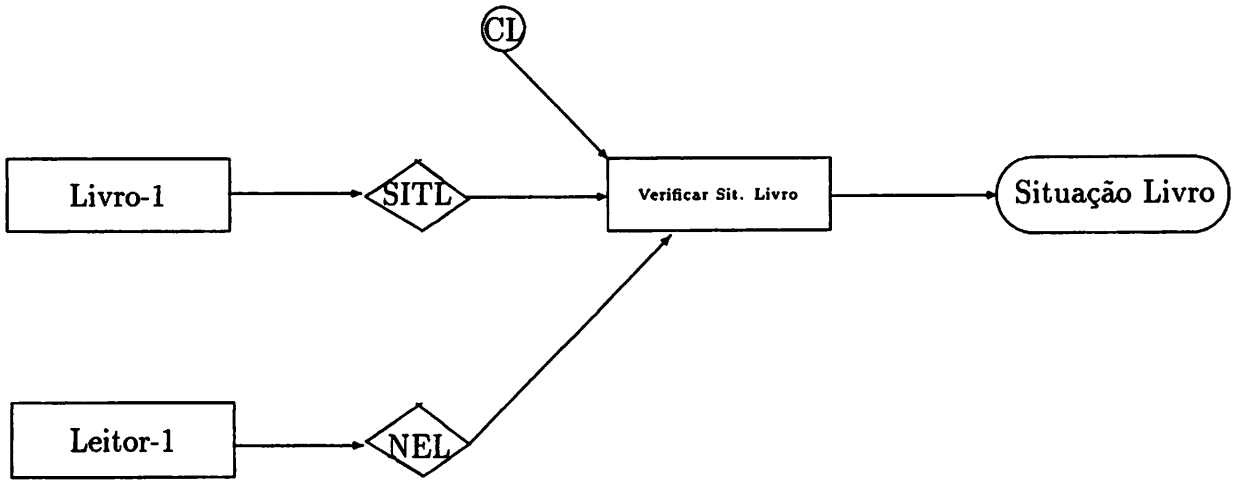
C.4 Fase IV - Funções

1. Bibliotecária solicita relatório de todos os livros comprados até aquele instante do ano e o custo total para a aquisição.



CO dados de informação de Livro-1 para Listar Livros
FA marcador de intervalo de tempo (final de ano)

2. Dada a identificação de um livro, indicar sua situação, isto é, se está na estante ou emprestado e, nesse caso para quem.



SITL informações sobre o estado de Livro-1
NEL informações sobre o estado de Leitor-1

C.5 Fase V - Temporização

As questões de temporização do sistema para esse estudo de caso foram resolvidas em sua maioria nas fases anteriores através da sincronização dos processos pela troca de mensagens entre eles, pela escolha do tipo de intercalação entre as sequências de dados do sistema, pela introdução de marcadores de tempo, etc.

C.6 Fase VI - Implementação

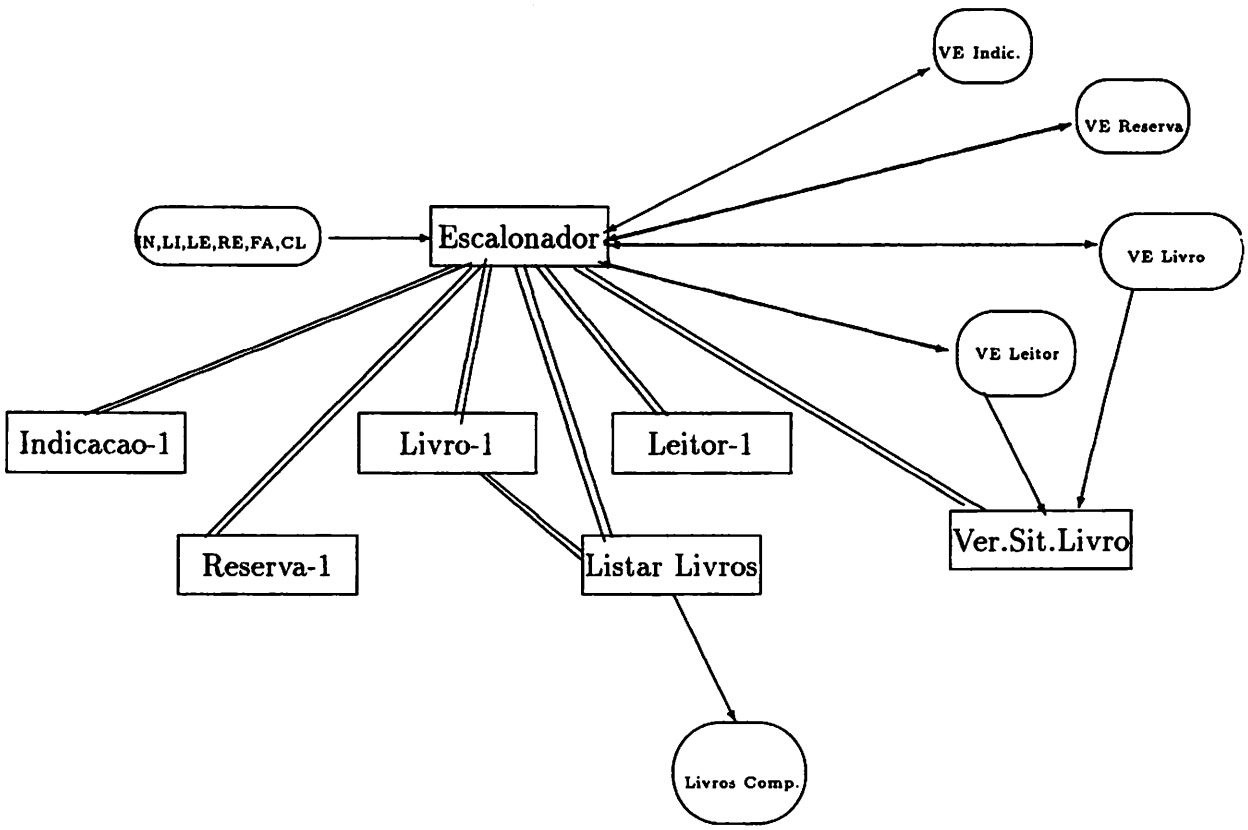


Figura C.5: Diagrama de Implementação do Sistema de Biblioteca

Apêndice D

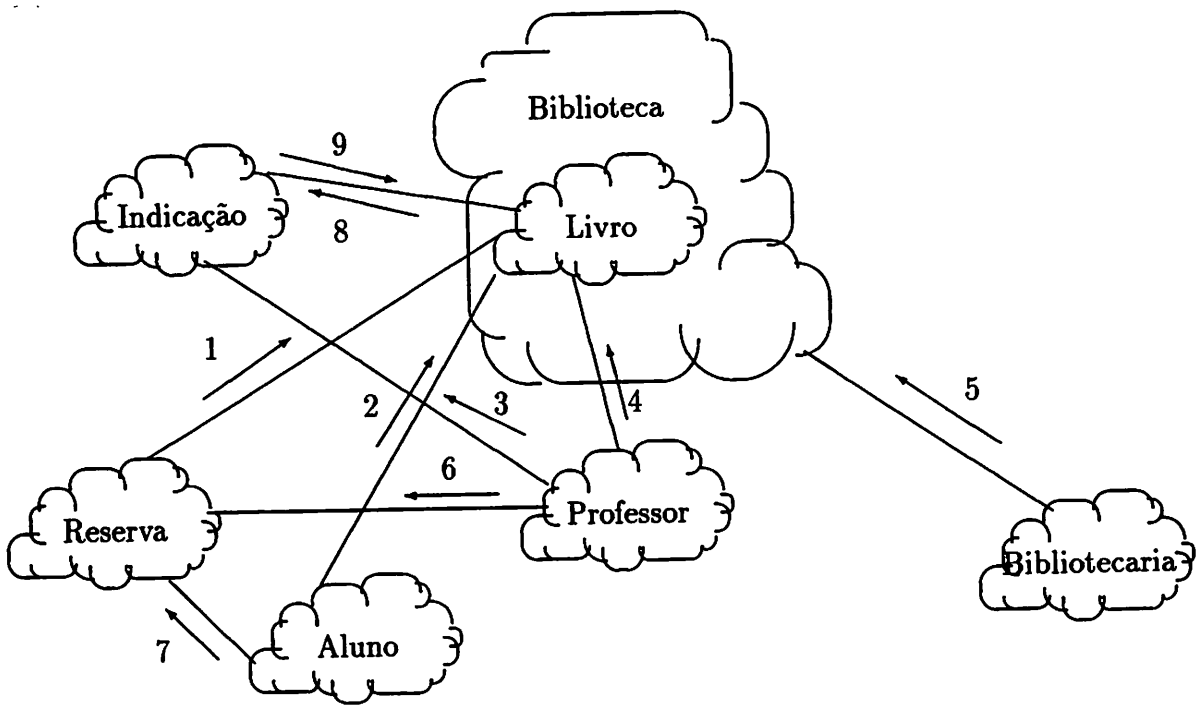
Estudo de Caso utilizando o Método OOD

Este apêndice tem como objetivo introduzir o leitor nos principais formalismos utilizados pelo método de Grady Booch [Bo91] através de um exemplo, sem entretanto utilizar toda notação disponível no OOD. O objetivo principal é mostrar o uso da notação diagramática com respeito a parte lógica do desenvolvimento de sistemas de informação, assim sendo, o restante do apêndice faz referência a modelagem do problema apresentado no estudo de caso desse trabalho, mostrando apenas os diagramas que dizem respeito à visão lógica do modelo. Isso se deve ao fato da adoção pelo JSD-OO, de parte da representação do projeto físico do método OOD, já descrita anteriormente.

O primeiro diagrama que normalmente o desenvolvedor constrói após a especificação inicial do problema, é o diagrama de objetos, que mostra um panorama instantâneo de interação dos objetos que formam o domínio do problema, surgidos a partir da especificação de requisitos do problema. A Figura D.1 mostra um dos diagramas de objetos que formam o modelo de objetos do problema apresentado no estudo de caso. O objeto **biblioteca** representa o conjunto de todos os livros que formam a biblioteca, e o objeto **bibliotecaria** representa uma aplicação do sistema, por exemplo, uma listagem dos livros mais recentemente adquiridos pela biblioteca e o seu custo de aquisição.

A seguir ou em paralelo à construção do diagrama de objetos inicial o projetista deve imaginar a estrutura de classes do problema, pensando nos mecanismos e nas abstrações chave que vão formar o conjunto de elementos do domínio da aplicação. Do diagrama de objetos anterior observa-se (sem pensar, nos objetos **bibliotecaria** e **biblioteca**) a necessidade de construção de cinco classes para instanciação dos objetos que formam o modelo de objetos em estudo: **indicação**, **reserva**, **professor**, **aluno** e **livro**. O projetista imagina o modelo, neste instante, de uma maneira Bottom-Up de construção, com as classes **professor** e **aluno** formando subclasses mais específicas, herdeiras da classe mais genérica **leitor**.

A Figura D.2 mostra a estrutura de classes através do diagrama de classes do modelo inicial com as cardinalidades dos relacionamentos de utilização sendo representadas. Pode-se observar neste diagrama o relacionamento de herança da classe **professor** representada



LEGENDA:

1 - Emprestar sob Reserva.
2 - Emprestar,Repor,Renovar, Devolver,Retirar,Retornar.
3 - Indicar,Desistir.
4 - idem 2.
5 - Listar.
6 - Reservar,Cancelar.
7 - idem 6.
8 - Receber.
9 - Comprar.

Figura D.1: Diagrama de Objeto

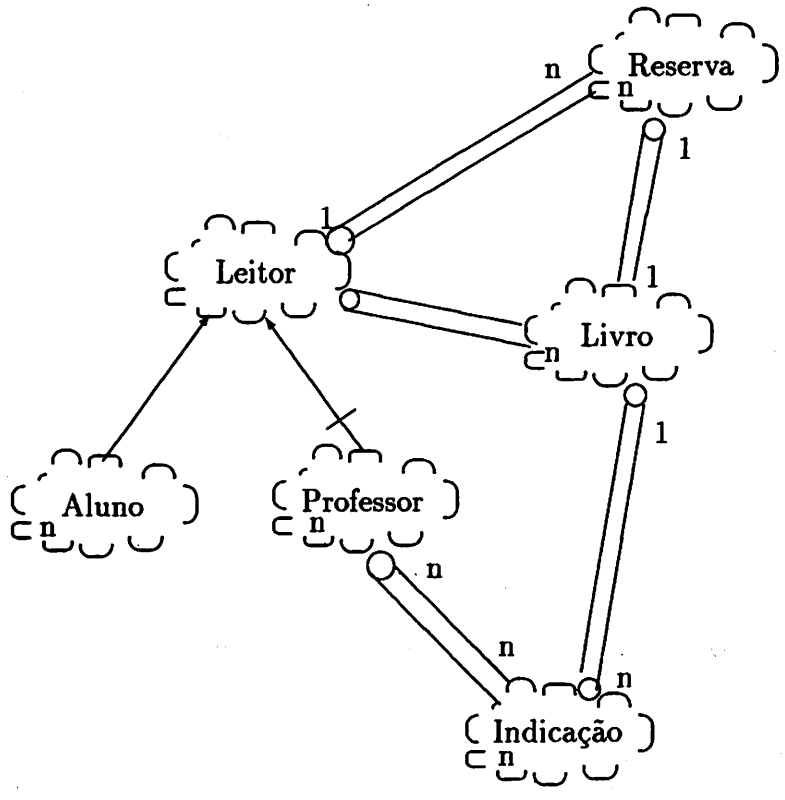


Figura D.2: Diagrama de Classes

por um linha orientada apontando para a classe da qual se herda, cortada por um risco indicando que essa classe modifica ou aumenta os métodos herdados.

Por último, apresentamos o diagrama de temporização com o objetivo principal de mostrar a ordem de execução das operações dos objetos que comunicam-se. Esse diagrama não deve necessariamente ser representado para todos os modos de interação entre operações dos objetos do sistema, mas apenas para aquelas onde os tempos de execução são mais críticos. A Figura D.3 mostra um instante de interação entre as operações definidas para quatro dos objetos do modelo inicial do sistema. As linhas pontilhadas representam aninhamento de operações.

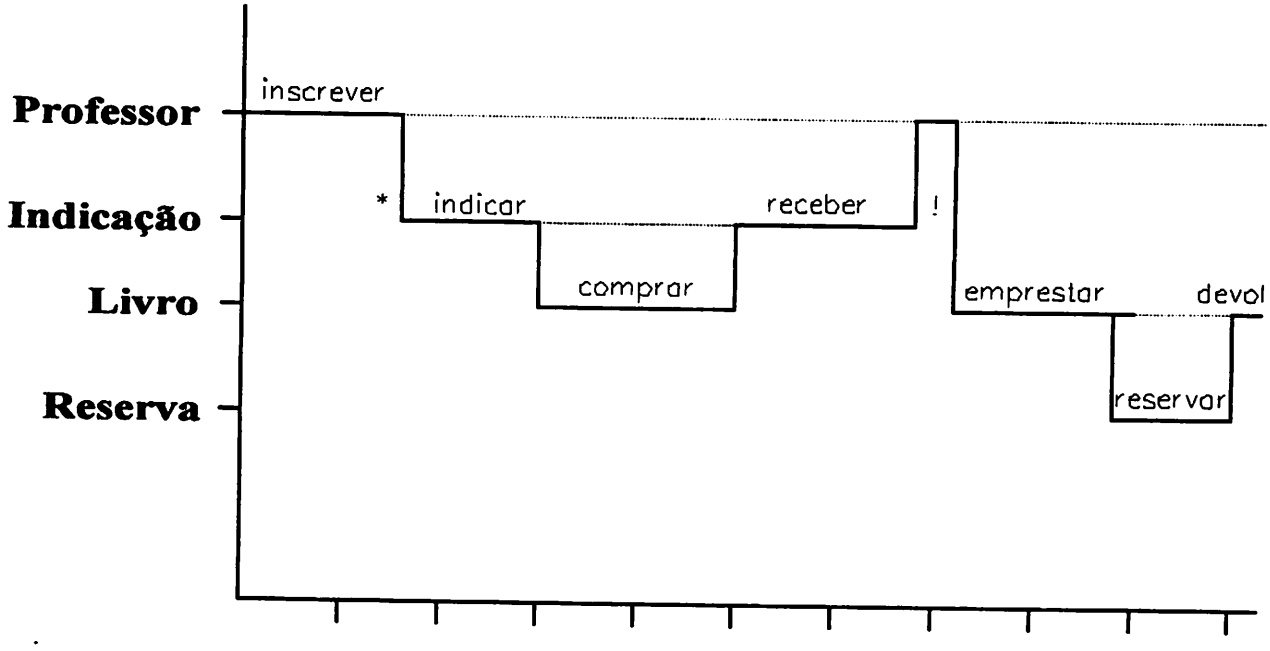


Figura D.3: Diagrama de Temporização

Bibliografia

[Al93] ALVES, R.M., BRAGA, J.L. Uma análise do método JSD - Jackson System Development - sob o ponto de vista de orientação a objetos, *Relatório Técnico RT004/93*, DCC, UFMG, Belo Horizonte, MG, março de 1993.

[Bo82] BOOCH, G. Object-oriented design, *Ada Letters*, 1(3), March/April 1982.

[Bo86] BOOCH, G. Object-oriented development, *IEEE Transactions on Software Engineering*, 12(2):211-221, February 1986.

[Bo91] BOOCH, G. *Object oriented design with applications*, Redwood City, California, The Benjamin/Cummings Publishing Company,Inc., 1991.

[Ca83] CAMERON, J.R. *JSP & JSD: The Jackson approach to software development*, IEEE Computer Society Press, New York, 1983.

[Ca86] CAMERON, J.R. An overview of JSD, *IEEE Transaction on Software Engineering*, SE(12):222-240, February 1986.

[Ch76] CHEN, P.P. The entity–relationship model: towards a unified view of data, *ACM Transactions on Database Systems*, 1(1):9-36, January 1976.

[Ch92] DE CHAMPEAUX, D., FAURE, P. A comparative study of object-oriented analysis methods, *Journal of Object-Oriented Programming*, pp 21-33, March/April 1992.

[Co91] COAD, P., YOURDON, E. *Object-oriented design*, Englewood Cliffs, NJ, Yourdon Press, Prentice Hall, 1991.

[Fi90] FISHER, A.S. *CASE: utilização de ferramentas para desenvolvimento de software*, Rio de Janeiro, RJ, Editora Campus, 1990.

[Gi90] GIRARDI, M. del R., PRICE, R.T. O paradigma de desenvolvimento por objetos, *Revista de Informática Teórica e Aplicada*, Porto Alegre, RS, 1(2):69-98, maio de 1990.

[Go84] GOLDBERG, A. *Smalltalk-80: the interactive programming environment*, Reading, Addison-Wesley, 1984.

[In91] INCE, D. *Object-oriented software engineering with C++*, Maidenhead, UK, The McGraw-Hill International (U.K.) Ltda, 1991.

[Ja75] JACKSON, M. *Principles of program design.* London, Academic Press, 1975.

[Ja78] JACKSON, M. Information systems: modelling, sequencing and transformations, *Proceedings of the Third International Conference on Software Engineering. IEEE*, 1978. Reimpresso por Bergland G.D. e Gordon, R.D. (EDS) Software Design Strategies (Tutorial), *IEEE Computer Society*, Catalog 389, pp 139-153, 1981.

[Ja81] JACKSON, M. Some principles underlying a system development method, *Systems Analysis and Design: A Foundation for 1980's*, North-Holland, W.W.Cotterman *et alli.*, eds, pp 185-198.

[Ja83] JACKSON, M. *System development*, London, Prentice-Hall, 1983.

[Ko90] KORSON, T., McGREGOR, J.D. Understanding object-oriented : a unifying paradigm, *Communications of the ACM*, 33(9):40-60, September 1990.

[Me87] MENEZES,C.S. *Notas de aula sobre o método JSD*, Manaus, AM, Universidade do Amazonas, 1987.

[Mo92] MONARCHI, D.E., PUHR, G.I. A research typology for object-oriented analysis and design, *Communications of the ACM*, 35(9):35-47, September 1992.

[Ms88] MASIERO, P.C., GERMANO, F.S.R. JSD as an object oriented design method, *ACM SIGSOFT*, Software Engineering Notes, 13(3):22-23, July 1988.

[Ms89] MASIERO, P.C. Uma visão geral do método JSD, *Notas Didáticas do ICMSC*, São Carlos, SP, USP, num.3, 1989.

[Ms92] MASIERO, P.C. *Análise estruturada de sistemas pelo método de Jackson*, São Paulo, SP, Editora Edgard Blücher Ltda., 1992.

[Ro89] RODRÍGUEZ, M.M.R. *Um analisador de especificações operacionais*, Dissertação de Mestrado, São Carlos, SP, ICMSC, USP, 1989.

[Ru91] RUMBAUGH, J. *et alli. Object-oriented modeling and design*, Englewood Cliffs, NJ, Prentice Hall, 1991.

[Se90] HENDERSON-SELLES, B., EDWARDS, J.M. The object-oriented systems life cycle, *Communications of the ACM*, 33(9):142-159, September 1990.

[Su88] SUTCLIFFE, A. *Jackson system development*, Hertfordshire, UK, Prentice Hall International (U.K) Ltd, 1988.

[Ve91] VERVLOET, W.J. *Um mapeamento de JSD para orientação a objetos*, série : Monografias em Ciência da Computação, 13/91, Rio de Janeiro, RJ, PUC-Rio, junho de 1991.

[We89] WEGNER, P. Learning the language, *BYTE*, pp 245-253, March 1989.

[Yo79] YOURDON, E., CONSTANTINE, L.L. *Structured design.* Englewood Cliffs, NJ, Prentice Hall, 1979.

[Yo89] YOURDON,E. *Modern structured analysis*, Englewood Cliffs, NJ, Prentice Hall, 1989.

[Za84] ZAVE,P. The operational versus the conventional approach to software development, *Communications of the ACM*, 27(2):104-118, February 1984.