

UFES - Universidade Federal do Espírito Santo

Análise de Sistemas

Notas de Aula

Ricardo de Almeida Falbo

E-mail: falbo@inf.ufes.br

2002/2

Índice

Capítulo 1 - Introdução	1
1.1 – Análise e Especificação de Requisitos	1
1.2 – A Organização deste Texto	2
PARTE I – ESPECIFICAÇÃO DE REQUISITOS	3
Capítulo 2 – Técnicas de Levantamento de Requisitos	4
2.1 – Amostragem	4
2.2 – Investigação	7
2.3 – Entrevistas	8
2.4 – Questionários	13
2.5 – Observação	18
2.6 – Prototipação	20
Capítulo 3 – Modelagem de Casos de Uso	23
3.1 – Casos de Uso	23
3.2 – Diagramas de Casos de Uso	25
3.3 – Descrição de Casos de Uso	26
3.4 – Associações entre Casos de Uso	28
PARTE II – ANÁLISE ORIENTADA A OBJETOS	33
Capítulo 4 – Introdução à Orientação a Objetos	34
4.1 – Abordagem Estruturada x Abordagem Orientada a Objetos	34
4.2 – Conceitos da Orientação a Objetos	36
4.3 – Processo de Desenvolvimento Orientado a Objetos	47
4.4 – A Linguagem de Modelagem Unificada	56
Capítulo 5 – Análise Orientada a Objetos	59
5.1 - Identificação de Classes	60
5.2 - Especificação de Hierarquias de Generalização / Especialização	62
5.3 - Identificação de Subsistemas	63
5.4 - Identificação de Associações e Definição de Atributos	64
5.5 - Determinação do Comportamento	69
5.6 - Definição das Operações	72
PARTE III – ANÁLISE ESSENCIAL DE SISTEMAS	75
Capítulo 6 – Introdução à Análise Essencial	76
6.1 - Conceitos	77
6.2 - Especificação da Essência do Sistema	82

Capítulo 7 – Modelagem de Dados	86
7.1 - Conceitos Básicos	86
7.2 - Restrições de Integridade ou Leis de Consolidação	90
7.3 - Outras Considerações sobre Atributos	94
7.4 - Outros Conceitos Importantes	96
7.5 - Dicionário de Dados	102
 Capítulo 8 – Modelagem Funcional	 104
8.1 - Conceitos Básicos	105
8.2 - Construindo DFDs	111
8.3 - Técnicas de Especificação de Processos	113

1 – Introdução

O desenvolvimento de software é uma atividade de crescente importância na sociedade contemporânea. A utilização de computadores nas mais diversas áreas do conhecimento humano tem gerado uma crescente demanda por soluções computadorizadas.

É importante observar que, associada ao acréscimo da demanda, a evolução do hardware tem sido mais acentuada, disponibilizando aos usuários máquinas cada vez mais velozes e com maior capacidade de processamento.

Neste contexto, identificou-se, já na década de 70, uma situação crítica no desenvolvimento de software, a chamada *Crise do Software* [Pressman00], caracterizada pelos seguintes fatos:

- demanda muito superior à capacidade de desenvolvimento;
- qualidade insuficiente dos produtos; e
- estimativas de custo e tempo raramente cumpridas nos projetos.

Visando melhorar a qualidade dos produtos de software e aumentar a produtividade no processo de desenvolvimento, surgiu a área de pesquisa denominada *Engenharia de Software*. A Engenharia de Software busca organizar esforços no desenvolvimento de ferramentas, metodologias e ambientes de suporte ao desenvolvimento de software.

Dentre as principais atividades de um processo de desenvolvimento de software, destaca-se a atividade de análise e especificação de requisitos, na qual os requisitos de um sistema são levantados e modelados, para só então ser projetada e implementada uma solução. Esta atividade é o objeto de estudo deste texto.

1.1 – Análise e Especificação de Requisitos

Um completo entendimento dos requisitos do software é essencial para o sucesso de um esforço de desenvolvimento de software. A atividade de análise e especificação de requisitos é um processo de descoberta, refinamento, modelagem e especificação. O escopo do software definido no planejamento do projeto é refinado em detalhe, as funções e o desempenho do software são especificados, as interfaces com outros sistemas são indicadas e restrições que o software deve atender são estabelecidas. Modelos dos dados requeridos, do controle e do comportamento operacional são construídos. Finalmente, critérios para a avaliação da qualidade em atividades subsequentes são estabelecidos.

Os principais profissionais envolvidos nesta atividade são o engenheiro de software (muitas vezes chamado analista) e o cliente / usuário.

Neste texto, dividiremos a atividade de Análise e Especificação de Requisitos em duas outras com propósitos mais específicos, ainda que extremamente relacionadas:

- Elicitação de Requisitos: nesta atividade, os requisitos são capturados sob uma perspectiva dos usuários, isto é, os modelos gerados procuram definir as funcionalidades (requisitos funcionais) e restrições (requisitos não funcionais) que devem ser consideradas para atender às necessidades dos usuários;
- Análise: nesta atividade, são modelados as estruturas internas de um sistema capazes de satisfazer os requisitos identificados.

A etapa de Elicitação de Requisitos (ou Especificação de Requisitos) é independente de paradigma, uma vez que trata os requisitos do sistema sob uma perspectiva externa. Entretanto, a atividade de Análise, que modela as estruturas internas de um sistema, é completamente dependente do paradigma adotado no desenvolvimento. Assim, este texto é dividido em três partes:

- **PARTE I - ESPECIFICAÇÃO DE REQUISITOS:** trata do levantamento e da modelagem dos requisitos segundo uma perspectiva externa, independente de paradigma. Nesta parte, são discutidas técnicas para levantamento de requisitos e a técnica de modelagem de casos de uso, para modelagem dos requisitos funcionais de um sistema.
- **PARTE II - ANÁLISE ORIENTADA A OBJETOS:** apresenta os principais conceitos da orientação a objetos e a linguagem de modelagem unificada (UML) e explora a modelagem de análise segundo o paradigma de objetos.
- **PARTE III - ANÁLISE ESSENCIAL DE SISTEMAS:** apresenta os principais conceitos da análise essencial e discute a modelagem de análise segundo o método da análise essencial, que adota o paradigma estruturado.

1.2 - A Organização deste Texto

Este texto procura oferecer uma visão geral atividade de Análise e Especificação de Requisitos e contém, além desta Introdução, mais sete capítulos, divididos em três partes.

Na PARTE I, o capítulo 2 – *Técnicas de Levantamento de Requisitos* – apresenta as principais técnicas utilizadas na identificação dos requisitos de um sistema. O capítulo 3 – *Modelagem de Casos de Uso* – trata da modelagem de requisitos funcionais, utilizando a técnica de modelagem de casos de uso.

Na PARTE II, o enfoque recai sobre o paradigma orientado a objetos (OO). O capítulo 4 – *Introdução à Orientação a Objetos* – discute os principais conceitos da orientação a objetos e alguns aspectos relacionados com o processo de desenvolvimento OO, bem como introduz a Linguagem de Modelagem Unificada (*Unified Modeling Language* – UML). O capítulo 5 – *Análise Orientada a Objetos* – discute a modelagem de análise segundo o paradigma OO, utilizando os modelos propostos na UML.

Finalmente, na PARTE III, discute-se a Análise Essencial. O capítulo 6 – *Introdução à Análise Essencial* – apresenta os principais conceitos e modelos utilizados na Análise Essencial. O capítulo 7 – *Modelagem de Dados* – apresenta o modelo de Entidades e Relacionamentos. No capítulo 8 – *Modelagem Funcional* – o foco é a técnica de Diagramas de Fluxos de Dados.

PARTE I – ESPECIFICAÇÃO DE REQUISITOS

2 – Técnicas de Levantamento de Requisitos (Referência: [Kendall92])

Em todo desenvolvimento de software, um aspecto fundamental é a captura dos requisitos dos usuários. Para apoiar este trabalho, diversas técnicas podem ser utilizadas.

2.1 – Amostragem (Referência: Capítulo 4 [Kendall92])

Em um levantamento de requisitos, geralmente um engenheiro de software se depara com duas importantes questões:

- Entre os muitos relatórios, formulários e documentos gerados pelos membros de uma organização, quais deverão ser objeto de investigação?
- Pode haver um grande número de pessoas afetadas pelo sistema de informação proposto. Quais delas devem ser entrevistadas, observadas ou questionadas?

Servindo de base para todas as técnicas de levantamento de requisitos, entre elas investigação, entrevistas e observação, estão as decisões cruciais dizendo respeito a *o que* examinar e *quem* questionar ou observar. Estas decisões podem ser apoiadas por uma abordagem estruturada chamada *amostragem*.

Amostragem é o processo de seleção sistemática de elementos representativos de uma população. Quando os elementos selecionados em uma amostragem são analisados, pode-se assumir que esta análise revelará informações úteis acerca da população como um todo.

Por que usar amostragem?

- diminuir custos;
- acelerar o processo de levantamento de informações;
- eficiência: a informação tende a ser mais apurada, já que menos elementos podem ser analisados, mas estes podem ser analisados com mais detalhes;
- reduzir tendências.

O Processo da Amostragem

Há quatro passos que um engenheiro de software deve seguir para projetar uma boa amostra:

1. Determinar os dados a serem coletados ou descritos: Definir o que coletar e para que, isto é, que tipo de técnica de levantamento de informação será usado depois. Coletar dados irrelevantes representa perda de tempo.
2. Determinar a população a ser amostrada (o que / quem): No caso de documentos, definir quais documentos investigar e de que período / intervalo. No caso de pessoas, estabelecer a que nível da organização pertencem ou se são pessoas de fora.
3. Escolher o tipo da amostra.
4. Decidir sobre o tamanho da amostra.

Os dois primeiros passos dizem respeito ao contexto do desenvolvimento. Os dois últimos referem-se à técnica de amostragem propriamente dita e são detalhados a seguir.

Tipos de Amostra

Elementos da amostra são selecionados ...	não baseada em probabilidades	baseada em probabilidades
diretamente, sem restrições	de Conveniência	Randômica Simples
segundo um critério específico	Intencional	Randômica Complexa

- *Amostra de Conveniência*: irrestrita, não utiliza probabilidades, mais fácil e, geralmente, apresenta resultados irreais. Ex: aviso chamando os interessados a participar de uma reunião.
- *Amostra Intencional*: a escolha é feita com base em critérios pré-estabelecidos pelo engenheiro de software, sem levar em conta probabilidades. É uma amostra apenas moderadamente confiável. Ex: engenheiro de software escolhe, para entrevista, um grupo de indivíduos que aparentam ter conhecimento e interesse no novo sistema.
- *Amostra Randômica Simples*: é necessário ter em mãos uma lista da população a ser amostrada (documentos ou pessoas) para garantir que cada elemento tem igual chance de ser selecionado. Geralmente, não é prática, especialmente para documentos e relatórios.
- *Amostra Randômica Complexa*: pode ser de três tipos:
 - ✓ *Amostra sistemática*: é o tipo mais simples de amostragem que leva em conta probabilidades. Consiste em se pegar sempre o k-ésimo elemento da população. Pode introduzir tendências.
 - ✓ *Amostra Estratificada*: é a abordagem mais importante para um engenheiro de software. Identifica sub-populações e escolhe elementos dentre essas sub-populações. Muito útil quando se deseja usar diferentes técnicas de levantamento de informação para sub-grupos específicos. Ex: coletar informações de pessoas de diferentes níveis da organização.
 - ✓ *Amostra de Grupos*: consiste em selecionar um grupo para ser estudado. Ex: selecionar um ou duas filiais de uma organização, assumindo que espelham o comportamento de todas filiais.

Tamanho da Amostra

O tamanho da amostra depende substancialmente do custo envolvido e do tempo requerido para se proceder a investigação, entrevista ou questionário posteriormente. O cálculo do tamanho da amostra varia, ainda, em função do tipo de informação que se deseja obter.

Quando a informação desejada for quantitativa, há dois procedimentos básicos de cálculo, dependentes do tipo de informação que se deseja obter:

- *Percentuais*: quando se deseja saber proporções ou percentuais, por exemplo o percentual de pessoas em uma organização que pensa de um certo modo, o tamanho da amostra pode ser calculado da seguinte forma:
 1. Determinar o atributo a ser amostrado.
 2. Localizar onde pode ser achado.
 3. Estimar o percentual da população que tem o atributo (p).
 4. Considerar um intervalo de aceitação (i). Ex: $\pm 10\%$
 5. Escolher nível de confiabilidade (%) e procurar seu correspondente coeficiente de segurança na tabela 2.1 (z).
 6. Calcular o erro padrão: $\sigma_p = i / z$.
 7. Determinar o tamanho da amostra (n): $n = (p (1 - p) / \sigma_p^2) + 1$

Nível de Confiabilidade (%)	Coeficiente de Segurança (z)
99	2,58
98	2,33
97	2,17
96	2,05
95	1,96
90	1,65
80	1,28
50	0,67

Tabela 2.1 – Valores de Coeficiente de Segurança [Kendall92].

- *Valores*: quando se deseja saber quantidades (valores) reais, por exemplo o número de erros de preenchimento de um determinado formulário, o tamanho da amostra pode ser calculado da seguinte forma:
 1. Determinar a variável a ser amostrada.
 2. Localizar onde pode ser achada.
 3. Examinar a variável para se ter uma idéia acerca de sua magnitude e dispersão. Idealmente, deveria se conhecer a média e o desvio padrão (s). Contudo, é exatamente isso que normalmente se quer saber ao fazermos uma amostragem. Logo, é necessário fazer uma estimativa inicial, que será refinada com a amostragem.
 4. Considerar um intervalo de aceitação (i).
 5. Escolher nível de confiabilidade (%) e procurar seu correspondente coeficiente de segurança na tabela 1.1 (z).
 6. Calcular o erro padrão da média: $\sigma_x = i / z$.
 7. Determinar o tamanho da amostra (n): $n = (s / \sigma_x)^2 + 1$

Quando as informações a serem coletadas forem qualitativas, é melhor tentar obtê-las através de entrevistas. Entretanto, não há fórmulas mágicas para ajudar engenheiros de software a determinar quantas pessoas entrevistar em uma organização. Esta decisão deve ser baseada no tempo gasto para se proceder uma entrevista. Uma boa regra de bolso, independentemente do tamanho da organização, consiste em entrevistar pelo menos três pessoas em cada nível da organização e uma pessoa por área funcional.

2.2 – Investigação (Referência: Capítulo 4 [Kendall92])

Muitas vezes, algumas informações são difíceis de serem obtidas através de entrevistas ou observação. Tais informações revelam, tipicamente, um histórico da organização e sua direção. Nestes casos, devemos utilizar **investigação**, isto é, análise de documentos.

Através de investigação, podemos obter mais facilmente informações, tais como tipos de documentos e problemas associados, informação financeira e contextos da organização. Tais informações são difíceis de serem obtidas através de outras técnicas de levantamento de requisitos, tais como entrevistas ou observação.

Análise de Documentos Quantitativos

Documentos com formato pré-determinado, tais como relatórios e formulários, trazem informações muito úteis a um engenheiro de software. Estes documentos têm um propósito específico e um público-alvo.

Relatórios de desempenho, por exemplo, podem mostrar metas de uma organização, a distância em relação à meta e a tendência atual. Relatórios usados no processo de tomada de decisão mostram informações compiladas e podem incorporar algum conhecimento sobre a estratégia da organização.

Fichas (registros) provêem atualizações periódicas do que está ocorrendo no negócio. Um engenheiro de software pode inspecionar uma ficha para: (i) checar erros em quantidades e totais, (ii) procurar oportunidades de melhorar o desenho da ficha, (iii) observar número e tipos de transações e (iv) procurar instâncias onde a introdução de um sistema computadorizado pode simplificar o trabalho (cálculos, por exemplo).

Formulários, assim como fichas, são muito úteis para o levantamento de requisitos. Devem ser inspecionados tanto formulários oficiais quanto não oficiais em uso. Exemplares de formulários em branco devem ser coletados, procurando-se observar o tipo, propósito e o público alvo. Deve-se, ainda, verificar quem realmente recebe o formulário.

Ao se examinar formulários preenchidos, observar se: (i) há itens não preenchidos, (ii) há formulários nunca usados, (iii) há formulários não oficiais usados regularmente e (iv) os formulários são preenchidos pelas pessoas certas.

Na investigação de formulários preenchidos, é possível detectar problemas como: (i) a informação não flui como planejado, (ii) pontos de gargalo no processamento de formulários, (iii) trabalho duplicado desnecessariamente, e (iv) falta de visão do fluxo global da informação, isto é, porque um formulário é preenchido e quem o utilizará.

Análise de Documentos Qualitativos

Documentos sem formato pré-determinado, tais como memorandos, quadros de aviso e manuais, também são úteis para o levantamento de requisitos, uma vez que mostram como os membros de uma organização são engajados nos processos da mesma.

A análise de documentos qualitativos deve envolver as seguintes tarefas:

- Examinar documentos para identificar como os elementos da organização são referenciados e, assim, conhecer a organização.
- Identificar disputas (entre departamentos ou com outras empresas) e, assim, conhecer a política da organização.
- Identificar termos que aparecem repetidamente em documentos e caracterizem o que é “bom” ou “ruim” para a organização.
- Reconhecer a existência de senso de humor nos documentos, o que pode indicar o tipo dos membros da organização (por exemplo, conservadores, ...).

Ao analisar memorandos (inclusive os eletrônicos), dê preferência àqueles enviados para toda a organização. Observe quem enviou e quem recebeu. Memorandos, tipicamente fluem horizontalmente ou de cima para baixo e provêm uma idéia clara de valores, crenças e atitudes dos membros da organização.

Na investigação de sinais e quadros de aviso, procure por indícios que apontem a cultura da organização. Ex: Segurança em 1º Lugar.

Finalmente, ao analisar manuais e políticas organizacionais, procure identificar como as coisas devem funcionar, como as metas estratégicas da organização devem ser atingidas e verifique se estes passos estão sendo seguidos ou não.

Tanto na análise de dados qualitativos quanto de dados quantitativos, procure observar não só os documentos correntes, mas também documentos arquivados.

2.3 – Entrevistas (Referência: Capítulo 5 [Kendall92])

Uma **entrevista** de levantamento de informações é uma conversa direcionada com um propósito específico, que utiliza um formato “pergunta-resposta”. Os objetivos de uma entrevista incluem:

- obter as opiniões do entrevistado, o que ajuda na descoberta dos problemas-chave a serem tratados;
- conhecer os sentimentos do entrevistado sobre o estado corrente do sistema;
- obter metas organizacionais e pessoais; e
- levantar procedimentos informais.

Entrevista x Investigação

Fatos obtidos em uma investigação podem explicar o desempenho passado.

Metas projetam o futuro. Entrevistas são importantes para se determinar metas.

O Processo de uma Entrevista

Em uma entrevista, o engenheiro de software está, provavelmente, estabelecendo um relacionamento com uma pessoa estranha a ele. Assim, é importante que ele:

- construa, rapidamente, uma base de confiança e entendimento;
- mantenha o controle da entrevista;
- venda a “idéia do sistema”, provendo ao entrevistado as informações necessárias.

Uma entrevista envolve as seguintes etapas principais: planejamento, condução e elaboração de um relatório da entrevista.

2.3.1 - Planejamento

O planejamento de uma entrevista envolve os seguintes passos:

1. *Estudar material existente sobre os entrevistados e suas organizações.* Procure dar atenção especial à linguagem usada pelos membros da organização, procurando estabelecer um vocabulário comum a ser usado na elaboração das questões da entrevista. Este passo visa, sobretudo, otimizar o tempo despendido nas entrevistas, evitando-se perguntar questões básicas e gerais.
2. *Estabelecer objetivos.* De maneira geral, há algumas áreas sobre as quais um engenheiro de software desejará fazer perguntas relativas ao processamento de informação e ao comportamento na tomada de decisão, tais como fontes de informação, formatos da informação, frequência na tomada de decisão, estilo da tomada de decisão, etc.
3. *Decidir quem entrevistar.* É importante incluir na lista de entrevistados pessoas-chave de todos os níveis da organização afetados pelo sistema. A pessoa de contato na organização pode ajudar nesta seleção. Quando necessário, use amostragem.
4. *Preparar a entrevista.* Uma entrevista deve ser marcada com antecedência e deve ter uma duração entre 45 minutos e uma hora.
5. *Decidir sobre os tipos de questões e a estrutura da entrevista.* O uso de técnicas apropriadas de questionamento é o “coração” de uma entrevista.
6. *Decidir como registrar a entrevista.* Entrevistas devem ser registradas para que informações obtidas não sejam perdidas logo em seguida. Os meios mais naturais de se registrar uma entrevista incluem anotações e o uso de gravador.

Tipos de Questões

Questões podem ser de três tipos básicos:

- *Questões subjetivas:* permitem respostas “abertas”. Ex: O que você acha de ...? Explique como você ...?

Vantagens:

- ✓ Provêm riqueza de detalhes.
- ✓ Revelam novos questionamentos.
- ✓ Colocam o entrevistado a vontade.
- ✓ Permitem maior espontaneidade.

Desvantagens:

- ✓ Podem resultar em muitos detalhes irrelevantes.
- ✓ Perda do controle da entrevista.
- ✓ Respostas muito longas para se obter pouca informação útil.
- ✓ Podem dar a impressão de que o entrevistador está perdido, sem objetivo.
- *Questões objetivas*: limitam as respostas possíveis. Ex: Quantos ...? Quem ...? Quanto tempo ...? Qual das seguintes informações ...?

Vantagens:

- ✓ Ganho de tempo, uma vez que vão direto ao ponto em questão.
- ✓ Mantêm o controle da entrevista.
- ✓ Levam a dados relevantes.

Desvantagens:

- ✓ Podem ser maçantes para o entrevistado.
- ✓ Podem falhar na obtenção de detalhes importantes.
- ✓ Não constróem uma afinidade entre entrevistador e entrevistado.
- *Questões de aprofundamento*: permitem explorar os detalhes de uma questão. Podem ser subjetivas ou objetivas. Ex: Por que? Você poderia dar um exemplo? Como isto acontece?

Questões Objetivas x Subjetivas

	Subjetivas	Objetivas
Confiabilidade dos dados	Baixa	Alta
Uso eficiente do tempo	Baixo	Alto
Precisão dos dados	Baixa	Alta
Amplitude e profundidade	Alta	Baixa
Habilidade requerida do entrevistador	Alta	Baixa
Facilidade de análise	Baixa	Alta

Tabela 2.2 – Quadro Comparativo Questões Objetivas x Subjetivas.

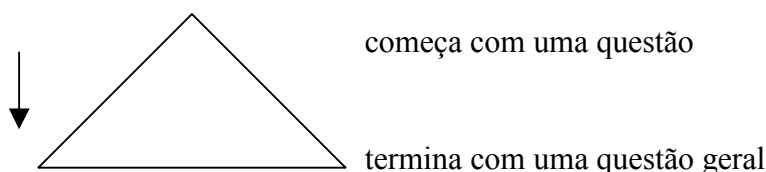
Problemas na Elaboração de Questões

- *Questões capciosas*: tendem a levar o entrevistado a responder de uma forma específica, isto é, são tendenciosas.
Ex: Sobre este assunto, você está de acordo com os outros diretores, não está?
Opção mais adequada: O que você pensa sobre este assunto?
- *Duas questões em uma*: O entrevistado pode responder a apenas uma delas, ou pode se confundir em relação à pergunta que está respondendo.
Ex: O que você faz nesta situação e como?

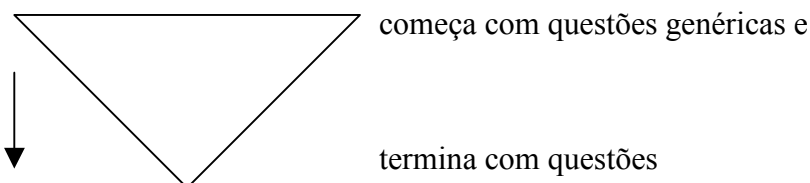
Estrutura da Entrevista

Diz respeito à organização das questões em uma sequência lógica. Há quatro formas básicas de se estabelecer a sequência das questões:

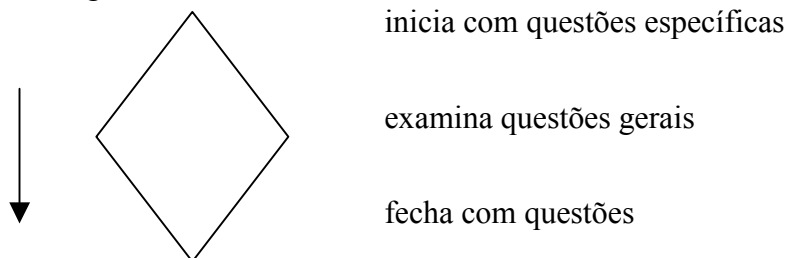
- *Estrutura de Pirâmide* (Abordagem Indutiva): inicia com questões bastante detalhadas, geralmente objetivas, e, à medida que a entrevista progride, questões mais gerais, subjetivas, são colocadas. Útil para situações onde o entrevistado parece relutante em abordar um assunto determinado ou se o engenheiro de software deseja obter uma finalização sobre o assunto.



- *Estrutura de Funil* (Abordagem Dedutiva): inicia com questões gerais subjetivas e, à medida que a entrevista avança, perguntas mais específicas, usando questões objetivas, são feitas. Esta estrutura provê um meio fácil e não ameaçador para se começar uma bateria de entrevistas. Permite levantar bastante informação detalhada, sendo desnecessárias longas seqüências de questões objetivas e de aprofundamento.



- *Estrutura de Diamante*: Combinação das duas anteriores: começa com questões específicas, passa a questões gerais e fecha a entrevista novamente com questões específicas. Frequentemente, é a melhor forma de se estruturar uma entrevista, já que mantém o interesse do entrevistado em uma variedade de questões. Contudo, tende a ser mais longa.



- *Entrevista Não Estruturada*: Não há uma definição da sequência das questões. De acordo com o andar da entrevista, caminhos possíveis são avaliados e a sequência é estabelecida. Requer mais tempo. Vale ressaltar que, ainda que a sequência das questões não seja definida a priori, as questões devem ser definidas antecipadamente, ou seja, o planejamento é necessário.

Entrevistas Estruturadas x Não Estruturadas

	Não Estruturada	Estruturada
Avaliação	Difícil	Fácil
Tempo Requerido	Alto	Baixo a Médio
Treinamento Requerido	Muito	Limitado
Espontaneidade	Alta	Baixa
“Insight” do Entrevistado	Muita Oportunidade	Pouca
Flexibilidade	Alta	Baixa
Controle	Baixo	Alto
Precisão	Baixa	Alta
Confiabilidade	Baixa a Média	Média a Alta
Amplitude e Profundidade	Alta	Baixa a Média

Tabela 2.3 – Comparação entre as Abordagens Estruturada e Não Estruturada.

Registro da Entrevista

É importante registrar os principais aspectos de uma entrevista durante a sua realização. No planejamento, deve-se definir como isto será feito. Há duas formas principais, cujas vantagens e desvantagens são apresentadas a seguir:

- *Gravador*: requer a permissão do entrevistado.

Vantagens:

- ✓ Registro completo da entrevista.
- ✓ Rapidez e melhor desenvolvimento.
- ✓ Reprodução para outros membros da equipe.

Desvantagens:

- ✓ Pode deixar o entrevistado pouco a vontade.
- ✓ Pode deixar o entrevistador distraído.
- ✓ Pode haver necessidade de transcrever a fita.

- *Anotações*

Vantagens:

- ✓ Mantém o entrevistador alerta.
- ✓ Pode ser usado para fornecer um roteiro para a entrevista.
- ✓ Mostra interesse e preparação do entrevistador.

Desvantagens:

- ✓ Perda do andamento da conversa.
- ✓ Excessiva atenção a fatos e pouca a sentimentos e opiniões.

2.3.2 – Condução da Entrevista

- Um dia antes, entre em contato com o entrevistado para confirmar o horário e o local da entrevista.
- Chegue um pouco antes do horário marcado.
- Apresente-se e esboçe brevemente os objetivos da entrevista.
- Relembre o entrevistado de que você estará registrando pontos importantes. Se for usar gravador, coloque-o em local visível.
- Diga ao entrevistado o que será feito com as informações coletadas e re-assegure seu aspecto confidencial.
- A entrevista deve durar entre 45 minutos e uma hora.
- Quando estiver incerto sobre uma questão, peça para o entrevistado dar definições ou outros esclarecimentos. Use questões de aprofundamento.
- Ao término da entrevista, pergunte se há algo mais sobre o assunto que o entrevistado ache importante você saber.
- Faça um resumo da entrevista e dê suas impressões globais.
- Informe o entrevistado sobre os passos seguintes.
- Pergunte se há outra pessoa com a qual você deveria conversar.
- Quando for o caso, marque nova entrevista.

2.3.3 – Relatório da Entrevista

O relatório ou ata da entrevista deve capturar a essência da entrevista. Escreva o relatório tão rápido quanto possível para assegurar qualidade.

Registre entrevistado, entrevistador, data, assunto e objetivos. Diga se os objetivos foram alcançados e aponte objetivos para entrevistas futuras. Registre, ainda, os pontos principais da entrevista e sua opinião.

2.4 – Questionários (Referência: Capítulo 6 [Kendall92])

O uso de questionários constitui uma técnica de levantamento de informações que permite ao engenheiro de software obter de várias pessoas afetadas pelo sistema (corrente ou proposto) informações, tais como:

- *Posturas*: o que as pessoas na organização dizem querer;
- *Crenças*: o que as pessoas pensam ser realmente verdade;
- *Comportamento*: o que as pessoas fazem;
- *Características*: propriedades de pessoas ou coisas.

Um questionário pode ter objetivos distintos, em função de sua aplicação, tais como:

- Procurar quantificar o que foi levantado em entrevistas.
- Determinar como um sentimento (expresso em uma entrevista) é realmente difundido ou limitado.
- Examinar uma grande amostra de usuários do sistema para sentir problemas ou levantar questões importantes, antes de se programar entrevistas.

Há muitas similaridades entre estas duas técnicas. De fato, pode ser útil utilizar as duas abordagens em conjunto:

- procurando refinar respostas não claras de um questionário em uma entrevista;
- projetando um questionário com base no que foi levantado em uma entrevista.

Questionários: Quando Usar?

- As pessoas estão espalhadas por toda a organização.
- Há um grande número de pessoas envolvidas no projeto do sistema e é necessário saber que proporção de um dado grupo aprova ou desaprova uma particular característica do sistema proposto.
- Em estudos exploratórios, quando se deseja saber uma opinião global antes de se definir qualquer direção específica para o projeto.

Etapas do Processo de Uso de Questionários

Assim como as entrevistas, para se empregar questionários, um conjunto de passos deve ser realizado, envolvendo pelo menos planejamento, aplicação e análise.

2.4.1 – Planejamento

No planejamento de um questionário, devem ser levados em consideração aspectos relacionados com a redação das questões, escalas, formato e ordem das questões.

Redação das Questões

Uma vez que questionários e entrevistas seguem uma abordagem “pergunta-resposta”, seria bastante razoável pensar que as considerações feitas para entrevistas aplicam-se também para questionários. Contudo, é importante ressaltar que há diferenças fundamentais entre estas técnicas e, portanto, novos aspectos devem ser considerados.

Em primeiro lugar, entrevistas permitem interação direta com o entrevistado a respeito das questões e seus significados. Em uma entrevista, o engenheiro de software pode refinar uma questão, definir um termo obscuro, alterar o curso do questionamento e controlar o contexto de modo geral. Isto não é necessariamente verdade para um questionário e, portanto, o planejamento de um questionário e de suas questões deve ser mais cuidadoso.

Um questionário deve:

- ter questões claras e não ambíguas,
- ter fluxo bem definido,

- ter administração planejada em detalhes, e
- levantar, antecipadamente, as dúvidas das pessoas que irão respondê-lo.

Questões Subjetivas

Quando for utilizar questões subjetivas em um questionário, antecipe o tipo de resposta que você espera obter. Estas questões devem ser restritas o suficiente para guiar as pessoas, de modo que respondam de uma maneira específica. Tome cuidado com perguntas que permitam respostas muito amplas, pois isto pode dificultar a comparação e a interpretação dos resultados.

Questões subjetivas devem ser usadas em questionários para levantar opiniões sobre algum aspecto do sistema ou em situações exploratórias.

Questões Objetivas

Questões objetivas devem ser utilizadas em um questionário:

- quando o engenheiro de software é capaz de listar as possíveis respostas ou
- para examinar uma grande amostra de pessoas.

Respostas a questões objetivas podem ser mais facilmente quantificadas. Respostas a questões subjetivas são analisadas e interpretadas de maneira diferente. A tabela 2.4 compara o uso de questões objetivas e subjetivas em questionários.

	Questões Subjetivas	Questões Objetivas
Tempo gasto para responder	Alto	Baixo
Natureza exploratória	Alta	Baixa
Amplitude e profundidade	Alta	Baixa
Facilidade de preparação	Alta	Baixa
Facilidade de análise	Baixa	Alta

Tabela 2.4 – Uso de questões subjetivas e objetivas em questionários.

Linguagem Utilizada: Diretrizes

- Sempre que possível, use o vocabulário das pessoas que irão responder. Prime pela simplicidade.
- Utilize perguntas simples e curtas.
- Evite redação tendenciosa.
- Garanta que as questões estão tecnicamente precisas antes de inclui-las no questionário.
- Para verificar a linguagem utilizada, aplique o questionário antecipadamente em um grupo piloto, pedindo atenção à adequabilidade dos termos empregados.

Escalas

São usadas para medir um atributo ou característica. A razão para se utilizar escalas é permitir medição ou julgamento. Escalas são geralmente arbitrárias e podem não ser únicas, por exemplo, temperatura: °C, °F, K. Há quatro tipos básicos de escalas:

- *Nominal*: utilizada para classificar coisas. É a forma mais “fraca” de medição, uma vez que só obtém totais para cada classificação.
Ex: Que tipo de software você mais usa?
1- Editor de Texto 2- Planilha 3- Gráfico 4- Outros
- *Ordinária*: também permite classificação, mas implica em um “rank”, isto é, uma escala é maior ou menor que a outra. Contudo, não se pode assumir que a distância entre as classes é a mesma.
Ex: O suporte técnico do Centro de Informação é:
(a) Extremamente útil (b) Muito útil (c) Útil
(d) Pouco útil (e) Nada útil
- *de Intervalo*: intervalos entre os números das opções são iguais, o que permite que sejam feitas operações matemáticas sobre os dados obtidos do questionário e, portanto, uma análise mais completa.
Ex: O suporte técnico do Centro de Informação é:
1- Nada útil 2 3 4 5- Extremamente útil
- *de Razão*: idem à de intervalo, só que possui um zero absoluto.
Ex: Quantas horas, aproximadamente, você despende diariamente no computador:
0 2 4 6 8

Tipos de Escala: Quando usar?

- de Razão: os intervalos são iguais e há um zero absoluto.
- de Intervalo: os intervalos são iguais, mas não há um zero absoluto.
- Ordinária: não é possível assumir que os intervalos são iguais, mas as classes podem ser colocadas em uma ordem.
- Nominal: deseja-se classificar coisas, mas estas não podem ser ordenadas.

Problemas na Construção de Escalas

- *Lenidade*: a pessoa responde a todas as questões do mesmo jeito. Solução: mover a categoria para a esquerda ou direita.
- *Tendência Central*: a pessoa responde tudo “na média”. Solução: criar uma escala com mais pontos, ajustar os descritores ou tornar as diferenças menores nos extremos.
- *Efeito “Auréola”*: a impressão formada em uma questão é levada para outra. Solução: mesclar questões sobre objetos diferentes.

Projeto do Questionário

Estilo

Um formulário bem projetado (aspectos de estilo) pode aumentar taxa de respostas. As seguintes diretrizes podem ser úteis na hora de se projetar um questionário:

- Deixe amplos espaços em branco para atrair as pessoas.
- Deixe espaço suficiente para as respostas das questões subjetivas.
- Em questões com escala, peça para fazer um círculo na resposta.
- Use os objetivos do questionário para ajudar a determinar o formato (inclusive instruções).
- Seja consistente no estilo. Coloque instruções sempre no mesmo local em relação ao lay-out do questionário, para facilitar a localização das instruções. Use letras maiúsculas e minúsculas nas perguntas e apenas letras maiúsculas nas respostas.

Ordem das Questões

Para ordenar as questões, considere os objetivos e, então, determine a função de cada questão para atingir esses objetivos. Use um grupo piloto para auxiliar ou observe o questionário com olhos de respondedor. Algumas orientações devem ser seguidas:

- As primeiras questões devem ser de interesse dos respondedores.
- Agrupe itens de conteúdo similar e observe tendências de associação.
- Coloque os itens de menor controvérsia primeiro.

2.4.2 – Aplicação do Questionário

A primeira questão a ser definida é: quem deve responder o questionário? A decisão de quem deve responder o questionário é feita em conjunto com o estabelecimento dos seus objetivos. Quando houver muitas pessoas aptas a responder o questionário, use amostragem.

Métodos de Aplicação

- Reunir todos os respondedores em um mesmo local para a aplicação do questionário.

Vantagens:

- ✓ 100% de retorno
- ✓ Instruções uniformes
- ✓ Resultado rápido

Problemas:

- ✓ Pode ser difícil reunir todas as pessoas.
- ✓ O respondedor pode ter coisas importantes a fazer.

- Analista entrega e recolhe cada questionário individualmente.

Vantagens:

- ✓ Boa taxa de resposta

Problemas:

- ✓ Desperdício do tempo do analista.
- ✓ O respondedor pode ser identificado.

- Respondedor administra o questionário.

Vantagens:

- ✓ Anonimato garantido.
- ✓ Respostas mais reais.

Problemas:

- ✓ Taxa menor de respostas. Este problema pode ser minimizado, mantendo-se uma lista de respondedores e controlando a devolução.

- Por correspondência. Útil somente para alcançar pessoas distribuídas geograficamente.

2.5 - Observação (Referência: Capítulo 7 [Kendall92])

Observar o comportamento e o ambiente do indivíduo que toma decisões pode ser uma forma bastante eficaz de levantar informações que, tipicamente, passam despercebidas usando outras técnicas.

Tomadas de decisão ocorrem em diversos níveis da organização: operacional, gerencial e estratégico e, portanto, é importante observá-las em todos os níveis que tenham interação com o sistema. Através da observação é possível capturar:

- o que realmente é feito e não apenas o que é documentado ou explicado.
- o relacionamento entre o “tomador de decisão” e outros membros da organização.

A observação é usada para:

- obter informações sobre o “tomador de decisão” e seu ambiente, que não são capturadas por outras técnicas.
- confirmar ou negar informações de entrevistas e/ou questionários.

Alguns pontos importantes devem ser realçados: o analista deve saber o que observar, quem observar, quando, onde, porque e como.

Observação do Comportamento

Permite observar como um gerente obtém, processa, compartilha e usa a informação para executar seu trabalho. No planejamento da observação do comportamento, os seguintes passos devem ser realizados:

1. Decidir o que observar (atividades).
2. Decidir em que nível de detalhe a atividade deve ser observada.
3. Preparar material para a observação.
4. Decidir quando observar
 - Amostragem de Horários: períodos para observação escolhidos aleatoriamente. Evita tendências, mas não permite a observação completa de um evento e tão pouco de um evento pouco freqüente.
 - Amostragem de Eventos: observação de eventos completos.

O ideal é combinar estas duas abordagens.

A observação deve ser registrada. Para tal, na preparação do material para a observação, as seguintes abordagens podem ser empregadas:

- Pares de adjetivos: estabeleça pares de adjetivos que capturem adequadamente o comportamento do indivíduo durante a tomada de decisão, tais como, decidido/ indeciso, confidencial/não confidencial, etc.
- Categorias: defina previamente categorias de atividades e durante a observação anote sua ocorrência ou não.
Ex: O Gerente instrui subordinados
 questiona subordinados
 lê informação externa
 processa informações
- Scripts: Para cada indivíduo observado, escreva uma lista de atividades por ele desempenhadas. O “tomador de decisão” é o ator, que é observado atuando, isto é, tomando decisões.

Observação de Ambiente Físico

O “tomador de decisão” influencia e é influenciado pelo seu meio físico. A observação do ambiente físico tem uma forte analogia com a crítica de filmes. Muitas vezes, é possível observar particularidades do ambiente físico que confirmam ou negam narrativas encontradas em entrevistas e questionários.

Uma forma sistemática de se proceder uma observação do ambiente físico é a chamada *observação estruturada do ambiente*. Ela é sistemática porque provê uma abordagem padrão para análise de elementos que influenciam a tomada de decisão, permitindo que vários engenheiros de software utilizem uma mesma base. Dentre os elementos a serem observados, destacam-se:

- Localização do escritório em relação a outros escritórios: Escritórios de fácil acesso tendem a aumentar a freqüência de interação e o fluxo de mensagens

informais. Grupos de escritórios encorajam o compartilhamento de informações. Escritórios de difícil acesso tendem a aumentar a frequência de mensagens orientadas a tarefas.

- Móveis e publicações em geral: revelam necessidade de informação interna ou externa.
- Outros: vestimentas, equipamentos, etc.

2.6 – Prototipação (Referência: Capítulo 8 [Kendall92])

A prototipação é uma técnica valiosa para se obter rapidamente informações específicas sobre *requisitos de informação* do usuário. Tipicamente, a prototipação permite capturar os seguintes tipos de informação:

- *Reações iniciais do usuário*: Como o usuário se sente em relação ao sistema em desenvolvimento? Reações ao protótipo podem ser obtidas através da observação, entrevistas, questionário ou relatório de avaliação.
- *Sugestões do usuário para refinar ou alterar o protótipo*: guiam o engenheiro de software na direção de melhor atender as necessidades dos usuários.
- *Inovações*: novas capacidades, não imaginadas antes da interação com o protótipo.
- *Informações para revisão de planos*: estabelecer prioridades e redirecionar planos.

Abordagens para a Prototipação

- *Protótipo não-operacional*: apenas as interfaces de entrada e saída são implementadas; o processamento propriamente dito não. É útil para avaliar certos aspectos do sistema quando a codificação requerida pela aplicação é custosa e a noção básica do que é o sistema pode ser transmitida pela análise de suas entradas e saídas.
- *Protótipo “arranjado às pressas”*: o protótipo possui toda a funcionalidade do sistema final, mas não foi construído com o devido cuidado e, portanto, sua qualidade e desempenho são deficientes.
- *Protótipo “primeiro de uma série”*: um sistema piloto é desenvolvido para ser avaliado antes de ser distribuído. Útil quando o sistema será implantado em vários locais diferentes.
- *Protótipo de características selecionadas*: apenas parte das características do sistema final são implementadas. O sistema vai sendo construído em partes: cada protótipo aprovado passa a ser um módulo do sistema.

Prototipação como Alternativa para o Ciclo de Vida no Desenvolvimento de Sistemas

Quando um modelo de ciclo de vida clássico (seqüencial linear) é utilizado, dependendo do tamanho do sistema, o tempo requerido para completar o ciclo pode ser muito grande e os requisitos dos usuários podem ser alterados, fazendo com que o sistema entregue não satisfaça as necessidades dos usuários.

Usando a prototipação como uma alternativa para o ciclo de vida de desenvolvimento de um sistema, é possível capturar mais rapidamente se os requisitos colocados sobre o software estão em conformidade com o requerido pelos usuários.

De fato, a rigor, a abordagem de protótipo de características selecionadas não deveria ser considerada prototipação, mas sim parte da estratégia de um desenvolvimento incremental ou evolutivo. Contudo, as outras três abordagens poderiam ser utilizadas em um desenvolvimento com ciclo de vida com prototipação.

Decidindo quando e que tipo de Prototipação usar

Considerar:

- Tipo do problema a ser resolvido (domínio do problema, tipo do sistema)
- Solução a ser apresentada pelo sistema (tecnologia a ser empregada – domínio da solução)
- Novidade (em termos de tecnologia e do domínio do problema)
- Complexidade (considerar clareza dos requisitos e tamanho do sistema)

Diretrizes para o Desenvolvimento de um Protótipo

- *Trabalhe com módulos gerenciáveis:* para fins de prototipação não é necessário e muitas vezes, nem desejável, construir um sistema completo.
- *Construa o protótipo rapidamente:* a construção de um protótipo na fase de análise e especificação de requisitos não pode consumir tempo em demasia, caso contrário perde sua finalidade. Para acelerar a construção, use ferramentas adequadas.
- *Modifique o protótipo em iterações sucessivas:* o protótipo deve ser alterado em direção às necessidades do usuário. Cada modificação requer uma nova avaliação.
- *Enfatize a interface com o usuário:* as interfaces do protótipo devem permitir que o usuário interaja facilmente com o sistema. Um mínimo de treinamento deve ser requerido. Sistemas interativos com interfaces gráficas são muito indicados à prototipação.

Usuários na Prototipação

Usuários são fundamentais na prototipação. Para capturar as reações dos usuários em relação ao protótipo, outras técnicas de levantamento de informação devem ser usadas em conjunto. Durante a experimentação do usuário com o protótipo, utiliza-se a observação. Para capturar opiniões e sugestões, podem ser empregados, além da observação, entrevistas e questionários.

Problemas da Prototipação

- *Gerência do projeto*: Normalmente, várias iterações são necessárias para se refinar um protótipo. Sob esta ótica, surge uma importante questão: quando parar? Se esta questão não for tratada com cuidado, a prototipação pode se estender indefinidamente. É importante, pois, delinear e seguir um plano para coletar, analisar e interpretar as informações de realimentação do usuário.
- *Considerar o protótipo como sendo o sistema final*: a qualidade pode não ter sido apropriadamente considerada.

Vantagens da Prototipação

- Permite alterar o sistema mais cedo no desenvolvimento, adequando-o mais de perto às necessidades do usuário (menor custo de uma alteração).
- Permite descartar um sistema quando este se mostrar inadequado (protótipo de viabilidade).
- Possibilidade de desenvolver um sistema que atenda mais de perto as necessidades e expectativas dos usuários. Permite uma interação com o usuário ao longo de todo o ciclo de vida do desenvolvimento.

Referências do Capítulo:

[Kendall92] K.E. Kendall, J.E. Kendall; *Systems Analysis and Design*, Prentice Hall, 1992.

3 – Modelagem de Casos de Uso

Quando um novo sistema precisa ser construído, surge uma importante questão: Como caracterizar os requisitos do sistema de um modo adequado para a Engenharia de Software, uma vez que, é necessário identificar quais os objetos/entidades relevantes, como eles se relacionam e como se comportam no contexto do sistema. Além disso, é preciso especificar e modelar o problema de maneira que seja possível criar um projeto efetivo.

O desenvolvimento de sistemas é um processo de construção de modelos, que tipicamente começa com um modelo de requisitos e termina com um modelo de implementação (código). Modelos de objetos (diagramas de classes, diagramas de interação, etc...) e modelos estruturados (diagramas de entidades e relacionamentos, diagramas de fluxo de dados, etc...) incluem detalhes, tais como, a estrutura interna dos objetos/entidades, suas associações, como eles interagem dinamicamente e como invocam o comportamento dos demais. Estas informações são necessárias para projetar e construir um sistema, mas não são suficientes para comunicar requisitos. Elas não capturam o conhecimento sobre as tarefas do domínio e, portanto, é difícil verificar se um modelo deste tipo realmente corresponde ao sistema que tem de ser construído [Jacobson96].

Assim, o primeiro modelo do sistema a ser construído deve ser passível de compreensão tanto por desenvolvedores - analistas, projetistas, programadores e testadores - como pela comunidade usuária - clientes e usuários. Modelos estruturados e de objetos são muito complexos e, portanto, não servem para este propósito. Este modelo inicial deve descrever o sistema, seu ambiente e como sistema e ambiente estão relacionados. Em outras palavras, ele deve descrever o sistema segundo uma perspectiva externa.

Modelos de caso de uso (*use cases*) são uma forma de estruturar esta visão externa. Como o próprio nome sugere, um *caso de uso é uma maneira de usar o sistema*. Usuários interagem com o sistema, interagindo com seus casos de uso. Tomados em conjunto, os casos de uso de um sistema representam tudo que os usuários podem fazer com este sistema. Casos de uso são os “itens” que um desenvolvedor vende a seus clientes.

Deve-se notar que modelos de caso de uso são independentes do método de análise a ser usado. Desenvolvedores devem modelar os casos de uso explicitamente bem no início do desenvolvimento de software. Em todo projeto, para que seja bem sucedido, casos de uso devem ser identificados [Jacobson96].

Em suma, o processo de desenvolvimento de software começa pelo entendimento de como o sistema será usado. Uma vez que as maneiras de usar o sistema tenham sido definidas, a modelagem pode, então, ser iniciada.

3.1 – Casos de Uso

Nenhum sistema computacional existe isoladamente. Todo sistema interage com atores humanos ou outros sistemas, que utilizam esse sistema para algum propósito e esperam que o sistema se comporte de acordo com as maneiras previstas. Um **caso de uso** especifica um comportamento de um sistema segundo uma perspectiva externa e é uma descrição de um conjunto de seqüências de ações realizadas pelo sistema para produzir um resultado de valor observável por um ator [Booch00].

Em essência, um **caso de uso** (*use case*) é uma interação típica entre um ator (usuário humano, outro sistema computacional ou um dispositivo) e um sistema. Um caso de uso captura alguma função visível ao ator e, em especial, busca atingir uma meta do usuário [Fowler97].

Os casos de uso fornecem uma maneira para os desenvolvedores chegarem a uma compreensão comum com os usuários finais do sistema e com os especialistas do domínio. Além disso, servem para ajudar a validar e verificar o sistema à medida que ele evolui durante seu desenvolvimento. Assim, os casos de uso não apenas representam o comportamento desejado do sistema, mas também podem ser utilizados como a base de casos de teste para o sistema, principalmente os testes de integração e de sistema [Booch00].

Casos de uso têm dois importantes papéis:

1. *Eles capturam os requisitos funcionais de um sistema.* Um modelo de caso de uso define o comportamento de um sistema (e a informação associada) através de um conjunto de casos de uso. O ambiente do sistema é definido pela descrição dos diferentes usuários. Estes usuários utilizam o sistema através de um número de casos de uso.
2. *Eles oferecem uma abordagem para a modelagem de sistemas.* Para gerenciar a complexidade de sistemas reais, é comum apresentar os modelos do sistema em um número de diferentes visões. Em uma abordagem guiada por casos de uso, pode-se construir uma visão para cada caso de uso, isto é, em cada visão são modelados apenas aqueles elementos que participam de um caso de uso específico. Um particular elemento pode, é claro, participar de vários casos de uso. Isto significa que um modelo do sistema completo só é visto através de um conjunto de visões - uma por caso de uso. Encontra-se todas as responsabilidades de um elemento de modelo, olhando todos os casos de uso onde este tem um papel.

Além de serem uma ferramenta essencial na captura dos requisitos de um sistema, casos de uso têm um papel fundamental no planejamento e controle de projetos iterativos.

A captura dos casos de uso é a primeira atividade a ser realizada no desenvolvimento, propriamente dito. A maioria dos casos de uso é normalmente gerada durante a fase de levantamento de requisitos, mas outros casos de uso podem ser descobertos à medida que o trabalho prossegue. Todo caso de uso é um requisito potencial e, enquanto um requisito não é capturado, não é possível planejar como tratá-lo.

Usualmente, em primeiro lugar, casos de uso são listados e discutidos, para só então, se realizar alguma modelagem. Entretanto, em alguns casos, a modelagem conceitual ajuda a descobrir casos de uso.

Um caso de uso pode ser capturado através de conversas com usuários típicos, discutindo as várias coisas que eles querem fazer com o sistema. Cada uma dessas interações discretas constitui um caso de uso. Dê a ela um nome e escreva uma descrição textual pequena (não mais do que uns poucos parágrafos). Não tente capturar todos os detalhes de um caso de uso logo no início.

Os objetivos do usuário podem ser o ponto de partida para a elaboração dos casos de uso. Proponha um caso de uso para satisfazer cada um dos objetivos do usuário. A partir deles, estude as possíveis interações do usuário com o sistema e refine o modelo de casos de uso.

3.2 - Diagramas de Casos de Uso

Diagramas de casos de uso especificam as funcionalidades que um sistema tem de oferecer, segundo diferentes perspectivas dos usuários. Basicamente, um diagrama de casos de uso apresenta dois elementos: os *atores* e os *casos de uso*. Um **ator** é um papel que um usuário, outro sistema ou dispositivo desempenha com respeito ao sistema. **Casos de uso** representam funcionalidades requeridas externamente. Uma associação entre um ator e um caso de uso significa que estímulos podem ser enviados entre atores e casos de uso. Os atores poderão estar conectados aos casos de uso somente por meio de associações. A associação entre um ator e um caso de uso indica que o ator e o caso de uso se comunicam entre si, cada um com a possibilidade de enviar e receber mensagens [Booch00].

A figura 3.1 mostra a notação básica da Linguagem de Modelagem Unificada – UML [Booch00] para diagramas de casos de uso.

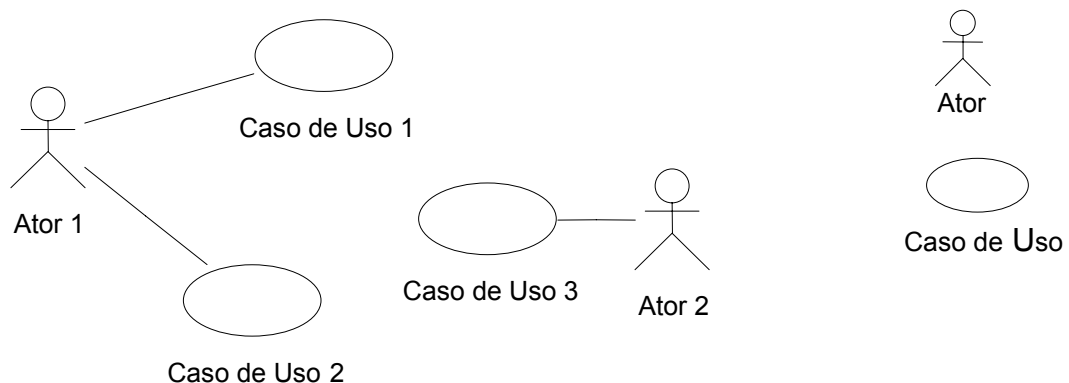


Figura 3.1 - Notação Básica da UML para Modelos de Caso de Uso.

Um ator modela qualquer coisa que precise interagir com o sistema, tais como usuários e outros sistemas que se comunicam com o sistema em questão. Atores são externos ao sistema; os casos de uso comportam os elementos de modelo que residem dentro do sistema. Assim, ao se definir fronteiras entre atores e casos de uso, está-se delimitando o escopo do sistema. Por estarem fora do sistema, atores estão fora do controle de nossas ferramentas de modelagem e não precisam ser descritos em detalhes. Atores representam tudo que tem necessidade de trocar informação com o sistema. Nada mais externo ao sistema deve ter impacto sobre o sistema.

É importante realçar a diferença entre ator e usuário. Um usuário é uma pessoa que utiliza o sistema, enquanto um ator representa um papel específico que um usuário pode desempenhar. Vários usuários em uma organização podem interagir com o sistema da mesma forma e, portanto, desempenham o mesmo papel. Um ator representa exatamente um certo papel que diversos usuários podem desempenhar. Assim, atores podem ser pensados como classes, isto é, descrições de um comportamento, enquanto usuários podem desempenhar diversos papéis e, assim, servir como instâncias de diferentes classes de atores. Ao lidar com atores, é importante pensar em termos de papéis ao invés de usuários. Um bom ponto de partida para a identificação de atores é verificar por que o sistema deve ser desenvolvido, procurando observar que atores o sistema se propõe a ajudar.

Quando um ator interage com o sistema, normalmente, ele realiza uma sequência comportamentalmente relacionada de ações em um diálogo com o sistema. Tal sequência compreende um *caso de uso*. Um caso de uso é, de fato, uma maneira específica de utilizar o sistema, através da execução de alguma parte de sua funcionalidade. Cada caso de uso constitui um curso completo de eventos com um ator e especifica a interação que acontece entre o ator e o sistema. O conjunto de todas as descrições de casos de uso especifica todas as maneiras de se usar o sistema e, conseqüentemente, a sua funcionalidade completa.

Um bom caso de uso compreende uma sequência de transações realizadas pelo sistema, que produz um *resultado de valor observável para um particular ator*. Por produzir um resultado de valor observável, queremos dizer que um caso de uso tem de garantir que um ator realiza uma tarefa que tem um valor identificável. Isso é importante para se obter casos de uso que não sejam muito pequenos. Por outro lado, a identificação de um particular ator é importante na obtenção de casos de uso que não sejam muito grandes.

Uma boa fonte para identificar casos de uso são os eventos externos. Pense sobre todos os eventos do mundo externo para os quais o sistema deve reagir. Identificar estes eventos pode ajudá-lo a identificar os casos de uso.

Para sistemas grandes, pode ser difícil propor uma lista de casos de uso. Nestas situações, é mais fácil chegar primeiro a uma lista de atores e tentar elaborar os casos de uso para cada ator. Para cada curso completo de eventos com um ator, um caso de uso é identificado. Nem sempre está claro qual funcionalidade deve ser colocada em casos de uso separados ou o que é apenas uma variante de um caso de uso. A priori, se as diferenças forem pequenas, elas podem ser descritas como variantes do caso de uso; caso contrário, devem ser descritas como casos de uso separados.

Qualquer que seja a abordagem utilizada, devemos sempre identificar os atores de um caso de uso, descobrir qual é o objetivo real do usuário e considerar modos alternativos de satisfazer estes objetivos.

3.3 - Descrição de Casos de Uso

Um caso de uso deve descrever *o que* um sistema faz. Geralmente, um diagrama de casos de uso é insuficiente para este propósito. Assim, deve-se especificar o comportamento de um caso de uso pela descrição textual de seu fluxo de eventos, de modo que alguém de fora possa compreendê-lo. Ao escrever o fluxo de eventos, deve-se incluir como e quando o caso de uso inicia e termina, quando o caso de uso interage com os atores e outros casos de uso e quais são as informações transferidas e o fluxo básico e fluxos alternativos do comportamento [Booch00].

Uma vez que o conjunto inicial de casos de uso estiver estabilizado, cada um deles deve ser descrito em mais detalhes. Primeiro, deve-se descrever o fluxo de eventos principal (ou curso básico), isto é, o curso de eventos mais importante, que normalmente ocorre. Variantes do curso básico de eventos e erros que possam vir a ocorrer devem ser descritos em cursos alternativos. Normalmente, um caso de uso possui apenas um único curso básico, mas diversos cursos alternativos. Tomemos como exemplo, um caixa automático de banco, cujo diagrama de casos de uso é mostrado na figura 3.2.

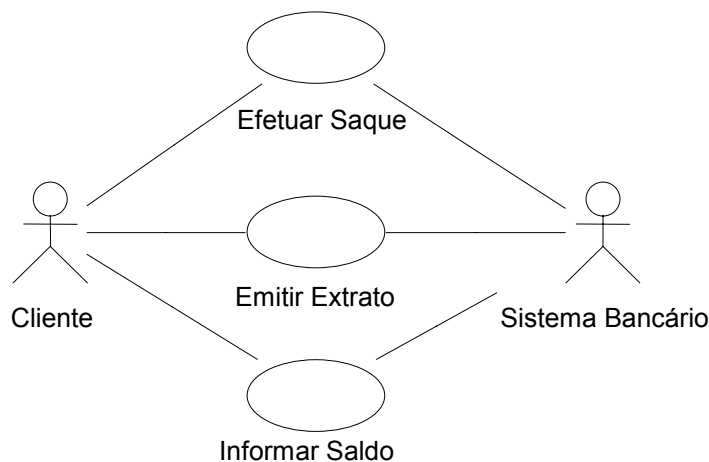


Figura 3.2 - Exemplo de um modelo de caso de uso.

O caso de uso *Efetuar Saque* poderia ser descrito da seguinte maneira:

Fluxo de Eventos Principal

Uma mensagem de saudação está sendo mostrada na tela. O cliente insere seu cartão no caixa automático, que lê o código da tarja magnética e checa se ele é aceitável.

Se o cartão é aceitável, o caixa pede ao cliente para informar a senha e fica aguardando até que ela seja informada.

O cliente informa a senha. Se a senha estiver correta, o caixa solicita que o cliente informe o tipo de transação e fica aguardando.

O cliente seleciona a opção saque. O caixa mostra uma tela para que seja informada a quantia. O cliente informa a quantia a ser sacada. O caixa envia uma requisição para o sistema bancário para que seja efetuado um saque na quantia especificada.

Se o saque é autorizado, as notas são preparadas para serem entregues, o cartão é ejetado, um recibo é emitido e as notas liberadas.

Cursos Alternativos

- O cartão não é aceitável: Se o cartão não é aceitável porque sua tarja magnética não é passível de leitura ou é de um tipo incompatível, o cartão é ejetado com um bip.
- Senha incorreta: Se a senha informada está incorreta, uma mensagem deve ser mostrada para o cliente que poderá entrar com a senha correta. Caso o cliente informe três vezes senha incorreta, o cartão deverá ser confiscado.
- Saque não autorizado: Se o saque não for aceito pelo Sistema Bancário, uma mensagem informando o cliente é mostrada por 10 segundos e o cartão é ejetado.
- Cancelamento: O cliente pode sempre cancelar a transação em qualquer momento que lhe seja perguntada alguma informação. Isto resultará na ejeção do cartão e no término da transação.

Como visto pelo exemplo anterior, um caso de uso pode ter um número de cursos alternativos que podem levar o caso de uso por diferentes fluxos. Tanto quanto possível, esses cursos alternativos, freqüentemente cursos de exceção, devem ser anotados durante a especificação de um caso do uso.

3.4 - Associações entre Casos de Uso

Uma vez que um modelo de caso de uso expressa apenas a visão externa do sistema, comunicações internas entre elementos dentro do sistema não devem ser modeladas. Contudo, para permitir uma descrição mais apurada dos casos de uso em um diagrama, três tipos de relacionamentos entre casos de uso podem ser empregados. Casos de uso podem ser descritos como versões especializadas de outros casos de uso (relacionamento de **generalização/especialização**); casos de uso podem ser incluídos como parte de outro caso de uso (relacionamento de **inclusão**); ou casos de uso podem estender o comportamento de um outro caso de uso básico (relacionamento de **extensão**). O objetivo desses relacionamentos é tornar um modelo mais compreensível, evitar redundâncias entre casos de uso e permitir descrever casos de uso em camadas.

O relacionamento de **inclusão** entre casos de uso significa que o caso de uso base incorpora explicitamente o comportamento de outro caso de uso. O local em que esse comportamento é incluído deve ser indicado na descrição do caso de uso base, através de uma referência explícita à chamada ao caso de uso incluído. Um relacionamento de inclusão é empregado quando há uma porção de comportamento que é similar ao longo de um ou mais casos de uso e não se deseja repetir a sua descrição. Para evitar redundância e assegurar reuso, extrai-se essa descrição e compartilha-se a mesma entre diferentes casos de uso. Um relacionamento de inclusão é representado por uma dependência, estereotipada como **incluir** (*include*), como mostra a figura 3.3.

No exemplo do caixa automático, podemos perceber que todos os três casos de uso têm em comum uma porção que diz respeito à transação inicial do cartão. Neste caso, um relacionamento **incluir** deve ser empregado, como mostra a figura 3.3.

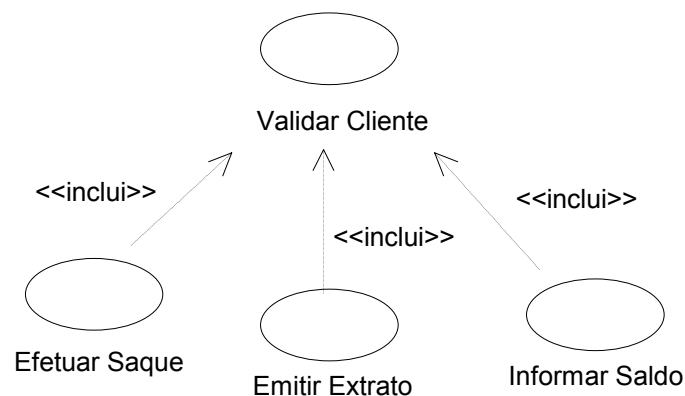


Figura 3.3 – O relacionamento **incluir** para descrever compartilhamento entre casos de uso.

O caso de uso *Validar* poderia ser descrito da seguinte forma:

Curso Normal

Uma mensagem de saudação está sendo mostrada na tela.

O cliente insere seu cartão no caixa automático, que lê o código da tarja magnética e checka se ele é aceitável.

Se o cartão é aceitável, o caixa pede ao cliente para informar a senha e fica aguardando até que ela seja informada.

O cliente informa a senha. Se a senha estiver correta, o caixa solicita que o cliente informe o tipo de transação e fica aguardando.

Cursos Alternativos

- O cartão não é aceitável: Se o cartão não é aceitável porque sua tarja magnética não é passível de leitura ou é de um tipo incompatível, o cartão é ejetado com um bip.
- Senha incorreta: Se a senha informada está incorreta, uma mensagem deve ser mostrada para o cliente que poderá entrar com a senha correta. Caso o cliente informe três vezes uma senha incorreta, o cartão deverá ser confiscado.
- Cancelamento: O cliente pode sempre cancelar a transação em qualquer momento que lhe seja perguntada alguma informação. Isto resultará na ejeção do cartão e no término da transação.

Com esta captura isolada do caso de uso de *Validar Cliente*, o caso de uso *Efetuar Saque* passaria a apresentar a seguinte descrição:

Curso Normal

O cliente realiza o caso de uso *Validar Cliente*.

O cliente seleciona a opção saque. O caixa mostra uma tela para que seja informada a quantia. O cliente informa a quantia a ser sacada. O caixa envia uma requisição para o sistema bancário para que seja efetuado um saque na quantia especificada.

Se o saque é autorizado, as notas são preparadas para serem entregues, o cartão é ejetado, um recibo é emitido e as notas liberadas.

Cursos Alternativos

- Saque não autorizado: Se o saque não for aceito pelo Sistema Bancário, uma mensagem informando o cliente é mostrada por 10 segundos e o cartão é ejetado.
- Cancelamento: O cliente pode sempre cancelar a transação em qualquer momento que lhe seja perguntada alguma informação. Isto resultará na ejeção do cartão e no término da transação.

O relacionamento de **generalização** entre casos de uso significa que o caso de uso filho herda o comportamento e o significado do caso de uso pai. O caso de uso filho deverá acrescentar ou sobrescrever o comportamento de seu pai e poderá ser substituído em qualquer local no qual o pai apareça [Booch00].

Voltando ao exemplo do sistema de caixa automático, suponha que haja duas formas adotadas para se validar o cliente: a primeira através de senha, como descrito anteriormente, e a segunda por meio de análise da retina do cliente. Neste caso, poderiam ser criadas duas especializações do caso de uso *Validar Cliente*, como mostra a figura 3.4.

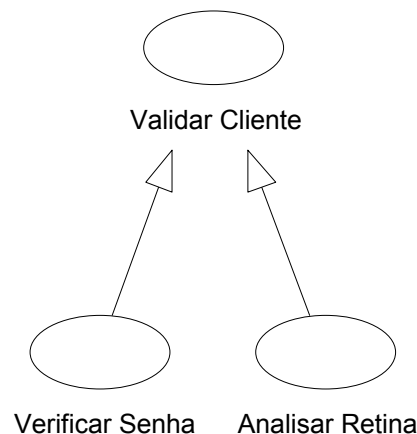


Figura 3.4 – O relacionamento de generalização/especialização entre casos de uso.

Um relacionamento de **extensão** entre casos de uso significa que o caso de uso base incorpora implicitamente o comportamento de um outro caso de uso em um local especificado em sua descrição como sendo um ponto de extensão. O caso de uso base poderá permanecer isolado, mas, sob certas circunstâncias, seu comportamento poderá ser estendido pelo comportamento de um outro caso de uso [Booch00].

Um relacionamento de extensão é utilizado para modelar uma parte de um caso de uso que o usuário considera como opcional. Desse modo, separa-se o comportamento opcional do comportamento obrigatório. Um relacionamento de extensão também poderá ser empregado para modelar um subfluxo separado, que é executado somente sob determinadas condições. Assim como o relacionamento de inclusão, o relacionamento de extensão é representado como uma associação de dependência, agora estereotipada como **estende** (*extend*).

O relacionamento **estende** nos permite capturar os requisitos funcionais de um sistema complexo de forma incremental. Inicialmente, as funções básicas são entendidas, para só então a complexidade ser introduzida. Na modelagem de casos de uso, devemos primeiro descrever os casos de uso básicos, para só então estendê-los para permitir descrever casos de uso mais avançados. O caso de uso no qual a nova funcionalidade é adicionada deve ter um curso de eventos completo e significativo por si próprio. Assim, sua descrição deve ser completamente independente.

Suponha que, no exemplo do caixa automático, deseja-se coletar dados estatísticos sobre o uso da transação de saque para alimentar o caixa eletrônico com as notas mais adequadas para saque. Para tal, poderíamos estender o caso de uso *Efetuar Saque*, de modo que, quando necessário, um outro caso de uso, denominado *Coletar Informações Estatísticas de Saque*, contasse e acumulasse o tipo das notas entregues em um saque. A figura 3.5 ilustra este caso.

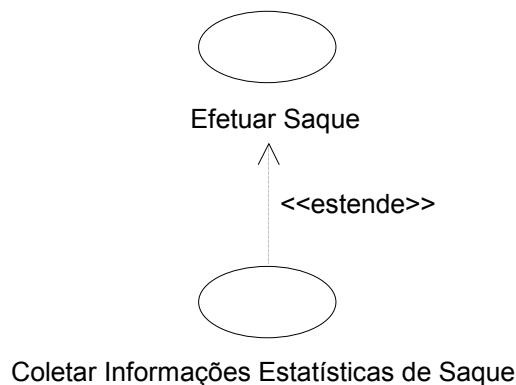


Figura 3.5 – O relacionamento **estende** entre casos de uso.

O relacionamento **estende** é usado, então, para modelar extensões a outros casos de uso completos e é utilizado para modelar:

- partes opcionais de casos de uso,
- cursos complexos e alternativos,
- subsequências que são executadas apenas em certos casos ou
- a inserção de diversos casos de uso diferentes dentro de um outro

A quebra de um caso de uso em vários pode ocorrer tanto na fase de levantamento de requisitos, quanto na fase de análise e projeto. Na fase de levantamento de requisitos, é interessante desmembrar aqueles casos de uso que se apresentam muito complicados. Entretanto, há casos de uso cuja complexidade global não deve ser detalhada antes da fase de projeto.

Para utilizar os relacionamentos de inclusão e extensão, aplique as seguintes regras:

- Utilize o relacionamento **estende** quando estiver descrevendo uma variação de um curso normal;
- Utilize o relacionamento **inclui** quando houver repetição em dois ou mais casos de uso separados e for desejável evitar esta repetição.

Durante a análise e descrição detalhada dos casos de uso, o sistema é cuidadosamente estudado. Uma vez que, geralmente, os casos de uso enfocam uma particular funcionalidade, é possível analisar a funcionalidade total do sistema de maneira incremental. Deste modo, casos de uso para áreas funcionalmente diferentes podem ser desenvolvidos independentemente e, mais tarde, reunidos para formar um modelo de requisitos completo. Com isso, permite-se enfocar um problema de cada vez, abrindo caminho para um desenvolvimento incremental.

À medida que os modelos crescem, é importante dividi-los para facilitar a leitura e o entendimento. Dividir um sistema complexo em subsistemas é sempre uma boa opção. Para dividir e reunir casos de uso em grupos relacionados conceitual e semanticamente, a UML oferece como ferramenta de modelagem os pacotes.

A figura 3.6 mostra um exemplo no contexto de um sistema bancário. Neste exemplo, o sistema bancário não está mais sendo considerado um outro sistema, mas um subsistema do mesmo sistema do caixa automático. A associação de dependência entre os pacotes mostrada nessa figura indica que o pacote *Caixa Automático* solicita serviços do pacote *Contas*.

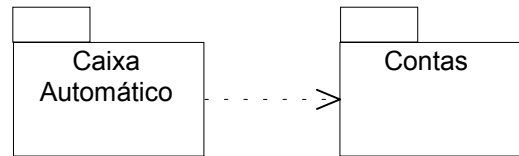


Figura 3.5 – Casos de Uso Agrupados em Pacotes.

Uma vez que a funcionalidade do sistema tiver sido capturada nos casos de uso, ela deve ser alocada a diferentes elementos de modelagem. Um elemento de modelagem, obviamente, pode ser comum a diversos casos de uso. Basicamente, o mapeamento de um caso de uso em elementos de modelo pode ser apoiado pelos seguintes princípios:

- As funcionalidades dos casos de uso que lidam com o registro e a manipulação de informação são mapeadas diretamente para elementos em um modelo de análise.
- As funcionalidades diretamente dependentes do ambiente do sistema representam funcionalidades requeridas pela interface do sistema e, portanto, são tratadas na fase de projeto, especificamente no projeto da interface do sistema.
- Finalmente, funcionalidades que tipicamente envolvem o controle de uma aplicação devem ser mapeadas em elementos no projeto do controle do sistema.

Referências do Capítulo:

- [Booch00] G. Booch, J. Rumbaugh, I. Jacobson; *UML – Guia do Usuário*. Editora Campus, 2000.
- [Fowler97] M. Fowler, K. Scott; *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley Object Technology Series, 1997.
- [Furlan98] J.D. Furlan; *Modelagem de Objetos Através da UML*; Makron Books, 1998.
- [Jacobson96] I. Jacobson; “The Use Case Construct in Object-Oriented Software Engineering”, In: *Scenario-Based Design*, 1996.

PARTE II – ANÁLISE ORIENTADA A OBJETOS

4. Introdução à Orientação a Objetos

A construção de uma solução computadorizada consiste no mapeamento do problema a ser resolvido no mundo real num modelo de solução no Espaço de Soluções, isto é, o meio computacional. A modelagem envolve, então, a identificação de objetos e operações relevantes no mundo real e o mapeamento desses em representações abstratas no Espaço de Soluções.

À distância existente entre o problema no mundo real e o modelo abstrato construído, convencionou-se chamar *gap semântico* e, obviamente, quanto menor ele for, mais direto será o mapeamento e, portanto, mais rapidamente serão construídas soluções para o problema. Sob essa ótica, é fácil perceber que o *gap semântico* representa a área de atuação da Engenharia de Software. Diversas técnicas e métodos têm sido propostos para as diferentes fases do processo de desenvolvimento, buscando minimizá-lo. A Orientação a Objetos é um dos paradigmas existentes para apoiar o desenvolvimento de sistemas, que busca fornecer meios para se diminuir o *gap semântico*. Este capítulo visa introduzir os principais conceitos e benefícios da orientação a objetos.

4.1 – Abordagem Estruturada x Abordagem Orientada a Objetos

Uma vez que, atualmente, a Orientação a Objetos tem tomado o espaço antes ocupado pelo paradigma estruturado no desenvolvimento de sistemas, é interessante fazer uma comparação entre os paradigmas que fundamentam estas abordagens:

- ❑ *Estruturado*: adota uma visão de desenvolvimento baseada em um modelo entrada-processamento-saída. No paradigma estruturado, os dados são considerados separadamente das funções que os transformam e a decomposição funcional é usada intensamente.
- ❑ *Orientado a Objetos*: pressupõe que o mundo é composto por *objetos*, onde um objeto é uma entidade que combina estrutura de dados e comportamento funcional. No paradigma orientado a objetos, os sistemas são estruturados a partir dos objetos que existem no domínio do problema, isto é, os sistemas são modelados como um número de objetos que interagem.

Em função do paradigma que os rege, métodos de análise e projeto de sistemas são classificados em *métodos estruturados* e *métodos orientados a objetos*.

Métodos Estruturados

Fazem clara distinção entre funções e dados. Funções, a princípio, são ativas e têm comportamento, enquanto dados são repositórios passivos de informação, afetados por funções. Esta divisão tem origem na arquitetura de hardware de von Neumann, onde a separação entre programa e dados é fortemente enfatizada.

Os *métodos orientados a funções* conduzem o desenvolvimento de software estruturando as aplicações segundo a ótica das funções (ações) que o sistema deverá realizar. O sistema é decomposto em funções, e os dados são transportados entre elas. Esta é a filosofia da proposta original da Análise Estruturada [DeMarco78] [Gane79], cuja ferramenta básica de modelagem são os diagramas de fluxo de dados (DFDs).

Os *métodos orientados a dados*, por sua vez, enfatizam a identificação e estruturação dos dados, subjugando a análise das funções para um segundo plano. Esses métodos têm origem no projeto de bancos de dados e, geralmente, têm no modelo de Entidades e Relacionamentos (ER) [Chen79] sua principal ferramenta.

A ênfase nas funções, geralmente, leva a sistemas com muita redundância e, conseqüentemente, inconsistentes e difíceis de serem integrados. Por outro lado, a ênfase nos dados está fundamentada em dois fatores significativos:

- Dados possuem existência própria nas organizações independentemente dos processos que os manipulam.
- Dados são muito mais estáveis que as funções em uma organização. A menos que haja grandes mudanças nos negócios de uma empresa, os dados tendem a se manter estáveis.

Assim, é possível desenvolver modelos de dados sem redundância, sem inconsistência e fáceis de integrar. Entretanto, uma vez que o modelo de dados deve representar a realidade, e o conhecimento da realidade, muitas vezes, passa pelo conhecimento das funções, ele deve ser construído de forma iterativa, não podendo ser considerado um produto acabado.

A Análise Essencial [Pompilho95] procurou conciliar as abordagens orientadas a dados e a funções em um único método, utilizando modelos para dados, funções e controles (DFDs e Modelo ER e Diagramas de Transição de Estados, respectivamente) como ferramentas para a modelagem de sistemas.

Um sistema desenvolvido usando um método estruturado, freqüentemente, é difícil de ser mantido. A princípio, o problema principal advém do fato de todas as funções terem de conhecer como os dados estão armazenados, isto é, a estrutura dos dados. Além disso, mudanças na estrutura dos dados quase sempre acarretam modificações em todas as funções relacionadas a essa estrutura. Em suma, a interpretação dos dados é apenas implícita, provida pelos programas que lêem ou escrevem dados. Diferentes programas podem dar diferentes interpretações aos dados e, portanto, é necessário conhecer como eles foram projetados para poder interpretá-los corretamente [Snyder93].

Métodos Orientados a Objetos

Os métodos orientados a objetos partem de um ponto de vista distinto e intermediário, onde se pressupõe que o mundo real é povoado por *objetos*, onde um objeto é uma entidade que combina estrutura de dados e comportamento funcional. Métodos orientados a objetos estruturam os sistemas a partir dos *objetos* que existem no domínio do problema.

A orientação a objetos oferece um número de conceitos bastante apropriados para a modelagem de sistemas. Utilizando a orientação a objetos como base, os sistemas são modelados como um número de objetos que interagem. Os modelos baseados em objetos são úteis para a compreensão de problemas, para a comunicação com os especialistas e usuários das aplicações, e para a realização das tarefas ao longo do ciclo de desenvolvimento de software. Os principais objetivos da orientação a objetos são:

- diminuir a distância conceitual entre o mundo real (domínio do problema) e o modelo abstrato de solução (domínio da solução);
- trabalhar com noções intuitivas (objetos e ações) durante todo o ciclo de vida, atrasando, ao máximo, a introdução de conceitos de implementação.

Normalmente, esta é uma maneira mais natural para descrever sistemas, já que os objetos são geralmente bastante estáveis. Alterações que por ventura venham a ocorrer, geralmente, afetam um ou alguns poucos objetos [Jacobson92].

Eduard Yourdon [Yourdon94] dá uma bom resumo do que pode ser considerado um produto “orientado a objeto”:

Um sistema construído usando um método orientado a objetos é aquele cujos componentes são partes encapsuladas de dados e funções, que podem herdar atributos e comportamento de outros componentes da mesma natureza, e cujos componentes comunicam-se entre si por meio de mensagens.

Métodos orientados a objetos utilizam uma perspectiva mais humana de observação da realidade, incluindo objetos, classificação e compreensão hierárquica. São benefícios esperados com o uso da orientação a objetos:

- Capacidade de enfrentar novos domínios de aplicação;
- Melhoria da interação entre analistas e especialistas;
- Aumento da consistência interna dos resultados da análise;
- Uso de uma representação básica consistente para a análise e projeto;
- Alterabilidade, legibilidade e extensibilidade;
- Possibilidade de ciclos de desenvolvimento variados;
- Apoio à reutilização.

É importante enfatizar, no entanto, que a orientação a objetos não é mágica, isto é, ela não é uma nova “tábua de salvação” para eliminar os problemas de produtividade e qualidade que têm atormentado a indústria de software ao longo das últimas décadas. Se praticada cuidadosamente, combinada com várias outras técnicas de Engenharia de Software - tais como, uso de métricas, reutilização, testes, e garantia da qualidade - a orientação a objetos pode ajudar a levar a melhorias substanciais no desempenho de uma organização de software [Yourdon94]. Portanto, é imprescindível, para grandes projetos, a definição de um processo de desenvolvimento que garanta o uso consistente dessas técnicas e que seja apoiado por ferramentas computacionais, tais como ferramentas CASE e Ambientes de Desenvolvimento de Software.

4.2 – Conceitos da Orientação a Objetos

4.2.1 - Princípios para Administrar Complexidade

O mundo real é algo extremamente complexo. Quanto mais de perto o observamos, mais claramente percebemos sua complexidade. A orientação a objetos tenta gerenciar a complexidade inerente dos problemas do mundo real, abstraindo conhecimento relevante e encapsulando-o dentro de objetos. De fato, alguns princípios básicos gerais para a administração da complexidade norteiam este processo de modelagem, entre eles, abstração, encapsulamento, modularidade e hierarquia.

Abstração

Uma das principais formas do ser humano lidar com a complexidade é através do uso de abstrações. As pessoas tipicamente tentam compreender o mundo, construindo modelos mentais de partes dele. Tais modelos são uma visão simplificada de algo, onde apenas

elementos relevantes são considerados. Modelos mentais, portanto, são mais simples do que os complexos sistemas que eles modelam.

Consideremos, por exemplo, um mapa como um modelo do território que ele representa. Um mapa é útil porque abstrai apenas aquelas características do território que se deseja modelar. Se um mapa incluísse todos os detalhes do território, provavelmente teria o mesmo tamanho do território e, portanto, não serviria a seu propósito.

Da mesma forma que um mapa precisa ser significativamente menor que o território que mapeia, incluindo apenas informações cuidadosamente selecionadas, um modelo mental abstrai apenas as características relevantes de um sistema para seu entendimento. Assim, podemos definir abstração como sendo o princípio de ignorar aspectos não relevantes de um assunto, segundo a perspectiva de um observador, tornando possível uma concentração maior nos aspectos principais do mesmo. De fato, a abstração consiste na seleção que um observador faz de alguns aspectos de um assunto, em detrimento de outros que não demonstram ser relevantes para o propósito em questão.

No que tange o desenvolvimento de software, duas formas adicionais de abstração têm grande importância: a abstração de dados e a abstração de procedimentos.

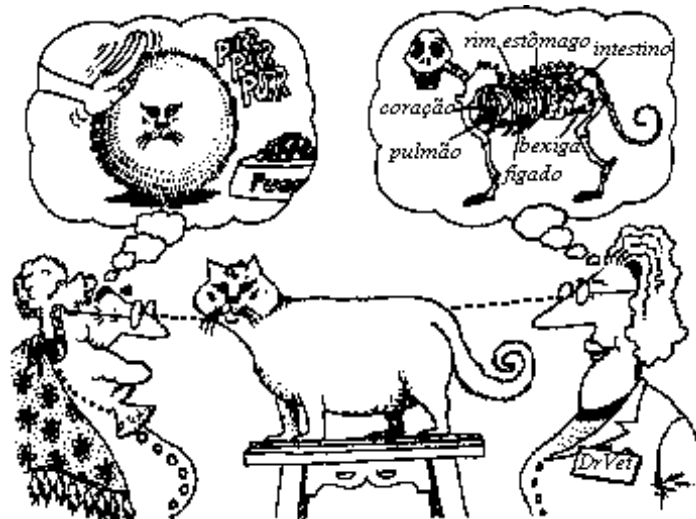


Figura 4.1 - A abstração enfoca as características essenciais de um objeto [Booch94].

Abstração de Dados

Consiste em definir um tipo de dado conforme as operações aplicáveis aos objetos deste tipo. Os objetos só podem ser modificados e observados através dessas operações [Coad92].

Exemplo: Um tipo de dado `pilha` pode ser definido através das operações `empilhar`, isto é, colocar um elemento no topo da pilha, e `desempilhar`, ou seja, retirar o elemento que está no topo da pilha. Um objeto do tipo `pilha` só pode ser modificado e observado através dessas duas operações.

Abstração de Procedimentos

Segundo esse princípio, uma operação com um efeito bem definido pode ser tratada por seus usuários como uma entidade única, mesmo que a operação seja realmente conseguida através de alguma seqüência de operações de nível mais baixo.

Exemplo: Seja a operação **calcular-salário-líquido** de um objeto do tipo **funcionário**. Essa operação pode ser tratada por seus usuários como uma entidade única, mesmo que ela seja, na realidade, construída como uma seqüência de operações de nível mais baixo, tais como: **calcular-INSS**, **calcular-IR**, **calcular-anuênio**, etc.

Encapsulamento

No mundo real, um objeto pode interagir com outro sem conhecer seu funcionamento interno. Uma pessoa, por exemplo, geralmente utiliza uma televisão sem saber efetivamente qual a sua estrutura interna ou como seus mecanismos internos são ativados. Para utilizá-la, basta saber realizar algumas operações básicas, tais como ligar/desligar a TV, mudar de um canal para outro, regular volume, cor, etc. Como estas operações produzem seus resultados, mostrando um programa na tela, não interessa ao telespectador.

O encapsulamento consiste na separação dos aspectos externos de um objeto, acessíveis por outros objetos, de seus detalhes internos de implementação, que ficam ocultos dos demais objetos [Rumbaugh94]. A interface de comunicação de um objeto deve ser definida de forma a revelar o menos possível sobre o seu funcionamento interno.

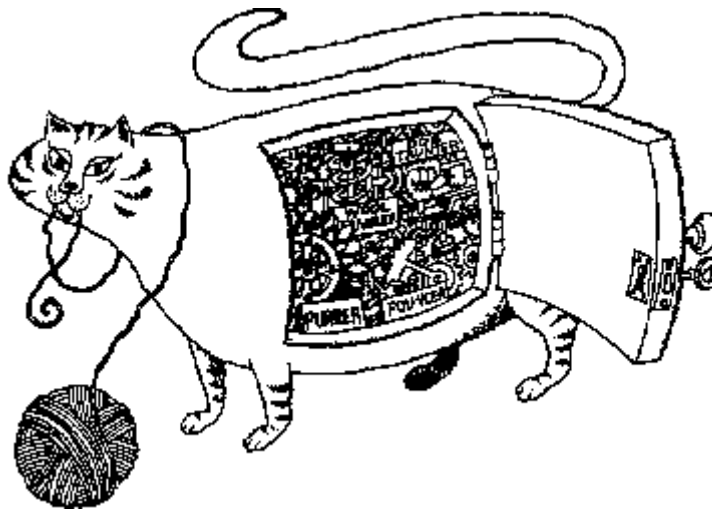


Figura 4.2 - O encapsulamento oculta os detalhes de implementação de um objeto [Booch94].

Abstração e encapsulamento são conceitos complementares: enquanto a abstração enfoca o comportamento observável de um objeto, o encapsulamento enfoca a implementação que origina esse comportamento. Encapsulamento é freqüentemente conseguido através do *ocultamento de informação*, isto é, escondendo detalhes que não contribuem para suas características essenciais. Tipicamente, em um sistema orientado a objetos, a estrutura de um objeto, e a implementação de seus métodos, são encapsuladas [Booch94].

Por exemplo, para usar um carro, uma pessoa não precisa conhecer sua estrutura interna (motor, caixa de marcha, etc...), nem tão pouco como se dá a implementação de seus métodos. Sabe-se que é necessário ligar o carro, mas não é preciso saber como esta operação é implementada. Assim, sobre carros, um motorista precisa conhecer apenas as operações que

permite utilizá-lo, a que chamamos de *interface do objeto*, o que inclui a ativação de operações, tais como ligar, mudar as marchas, acelerar, frear, etc..., e não como essas operações são de fato implementadas.

Encapsulamento serve para separar a interface contratual de uma abstração e sua implementação. Os usuários têm conhecimento apenas das operações que podem ser requisitadas e precisam estar cientes apenas do *que* as operações realizam e não *como* elas estão implementadas.

A principal motivação para o encapsulamento é facilitar a reutilização de objetos e garantir estabilidade aos sistemas. Um encapsulamento bem feito pode servir de base para a localização de decisões de projeto que necessitam ser alteradas. Uma operação pode ter sido implementada de maneira ineficiente e, portanto, pode ser necessário escolher um novo algoritmo. Se a operação está encapsulada, apenas o objeto que a define precisa ser modificado, garantindo estabilidade ao sistema.

Modularidade

Muitos métodos de construção de software buscam obter sistemas modulares, isto é, construídos a partir de elementos que sejam autônomos, conectados por uma estrutura simples e coerente. Modularidade é crucial para se obter reusabilidade e extensibilidade.

Modularidade é uma propriedade de sistemas decompostos em um conjunto de módulos coesos e fracamente acoplados. Assim, abstração, encapsulamento e modularidade são princípios sinérgicos¹. Um objeto provê uma fronteira clara em torno de uma abstração e o encapsulamento e a modularidade provêm barreiras em torno dessa abstração [Booch94].

Hierarquia

Abstração é um princípio importantíssimo, mas em todas as aplicações, exceto aquelas mais triviais, deparamo-nos com um número de abstrações maior do que conseguimos compreender em um dado momento. O encapsulamento ajuda a gerenciar esta complexidade através do ocultamento da visão interna de nossas abstrações. Modularidade auxilia também, dando-nos um meio de agrupar logicamente abstrações relacionadas. Entretanto, isto ainda não é o bastante. Um conjunto de abstrações frequentemente forma uma hierarquia e, pela identificação dessas hierarquias, é possível simplificar significativamente o entendimento sobre um problema [Booch94]. Em suma, hierarquia é uma forma de arrumar as abstrações.

4.2.2 - Conceitos Básicos

Objetos

O mundo real é povoado por elementos que interagem entre si, onde cada um deles desempenha um papel específico. A esses elementos, chamamos *objetos*. Objetos podem ser coisas concretas ou abstratas, tais como um carro, uma reserva de passagem aérea, uma organização, uma planta de engenharia, um componente de uma planta de engenharia, etc...

Do ponto de vista da modelagem de sistemas, um objeto é uma entidade que incorpora uma abstração relevante no contexto de uma aplicação. Um objeto possui um estado

¹ Sinergia: esforço simultâneo de vários elementos para a realização de uma ação.

(informação), exibe um comportamento bem definido, expresso por um número de operações para examinar ou alterar seu estado, e tem identidade única.

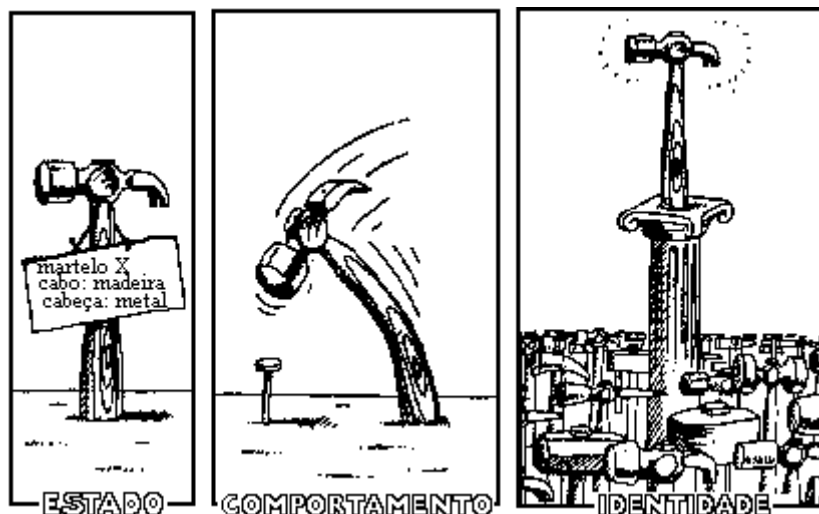


Figura 4.3 - Um objeto possui estado, exibe algum comportamento bem definido e possui identidade própria [Booch94].

Estado

O estado de um objeto compreende o conjunto de suas propriedades, associadas a seus valores correntes. Propriedades de objetos são geralmente referenciadas como atributos e, portanto, o estado de um objeto diz respeito aos seus atributos e aos valores a eles associados.

Comportamento

A abstração incorporada por um objeto é caracterizada por um conjunto de serviços ou operações, que outros objetos, ditos clientes, podem requisitar. Operações são usadas para recuperar ou manipular a informação de estado de um objeto e referem-se apenas às estruturas de dados do próprio objeto, não devendo acessar diretamente estruturas de outros objetos.

A comunicação entre objetos dá-se por meio de *troca de mensagens*. Para acessar a informação de estado de um objeto, é necessário enviar uma mensagem para ele. Uma mensagem consiste do nome de uma operação e os argumentos requeridos. Assim, o comportamento de um objeto representa como este objeto reage às mensagens a ele enviadas. Em outras palavras, o conjunto de mensagens a que um objeto pode responder representa o seu comportamento. Um objeto é, pois, uma entidade que tem seu estado representado por um conjunto de atributos (uma estrutura de informação) e seu comportamento representado por um conjunto de operações.

Identidade

Cada objeto tem uma identidade própria, que lhe é inerente. Todos os objetos têm existência própria, ou seja, dois objetos são distintos mesmo se seu estado e comportamento forem iguais. A identidade de um objeto transcende os valores correntes de suas variáveis de estado (atributos). Identificar um objeto diretamente é geralmente mais eficiente que designá-lo pela sua descrição [Snyder93].

No mundo real, um objeto limita-se a existir, mas, no que se refere ao mundo computacional, cada objeto dispõe de um identificador único pelo qual pode ser referenciado inequivocamente [Rumbaugh94].

Classes e Instâncias

É bastante comum encontrarmos no mundo real, diferentes objetos desempenhando um mesmo papel. Consideremos, por exemplo, duas cadeiras. Apesar de serem objetos diferentes, elas compartilham uma mesma estrutura e um mesmo comportamento. Entretanto, não há necessidade de se despendar tempo modelando as duas cadeiras, ou várias delas. Basta definir, em um único lugar, um modelo descrevendo a estrutura e o comportamento desses objetos. A esse modelo damos o nome de *classe*.

Uma classe descreve um conjunto de objetos com as mesmas propriedades (atributos), o mesmo comportamento (operações), os mesmos relacionamentos com outros objetos e a mesma semântica. Objetos que se comportam da maneira especificada pela classe são ditos *instâncias* dessa classe.

Todo objeto pertence a uma classe, ou seja, é instância de uma classe. De fato, a orientação a objetos norteia o processo de desenvolvimento através da *classificação de objetos*, isto é, objetos são agrupados em classes, em função de exibirem facetas similares, sem, no entanto, perda de sua individualidade. Assim, a modelagem orientada a objetos consiste, basicamente, na definição de classes. O comportamento e a estrutura de informação de uma instância são definidos pela sua classe.

Objetos com propriedades e comportamento idênticos são descritos como instâncias de uma mesma classe, de modo que a descrição de suas propriedades possa ser feita uma única vez, de forma concisa, independentemente do número de objetos que tenham tais propriedades em comum. Deste modo, uma classe captura a semântica das características comuns a todas as suas instâncias.

Enquanto um objeto individual é uma entidade real, que executa algum papel no sistema como um todo, uma classe captura a estrutura e comportamento comum a todos os objetos que ela descreve. Assim, uma classe serve como uma espécie de contrato que deve ser estabelecido entre uma abstração e todos os seus clientes.

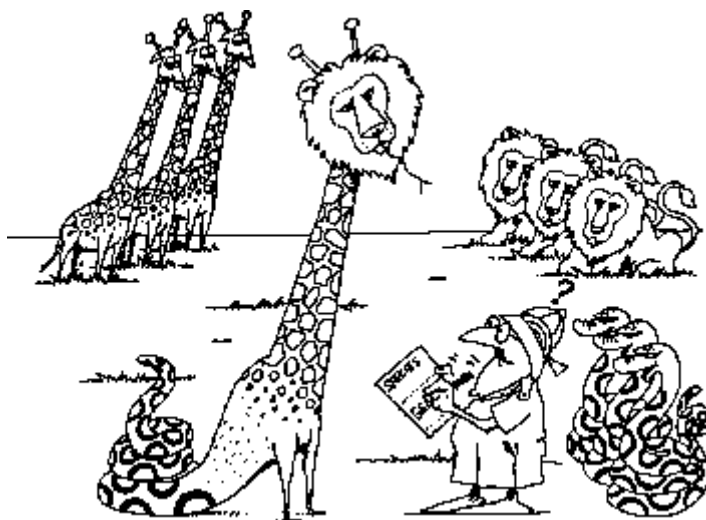


Figura 4.4 - Classificação é o meio pelo qual ordenamos conhecimento [Booch94].

Alguns autores utilizam os conceitos de classe e tipo indistintamente. Entretanto, um tipo e uma classe não são a mesma coisa. Um tipo é definido por um conjunto de operações, isto é, pelas manipulações que podemos fazer com o tipo. Uma classe é mais do que isso. Podemos também olhar para dentro de uma classe, por exemplo, para ver sua estrutura de informação. Assim, uma classe é melhor conceituada como uma implementação específica de um tipo [Jacobson92].

Mecanismos de Estruturação

Em qualquer sistema, objetos relacionam-se uns com os outros. Em sistemas não triviais, por sua vez, geralmente nos deparamos com uma razoável quantidade de objetos e classes e há necessidade de lidar com esta complexidade. Assim, é preciso estruturar classes e objetos. Vários mecanismos têm sido propostos, entre eles, associação, composição e herança.

Ligações e Associações

Objetos relacionam-se com outros objetos. Por exemplo, em “*o empregado João trabalha no departamento de Pessoal*”, temos um relacionamento entre o objeto `empregado João` e o objeto `departamento Pessoal`.

Ligações e associações são meios de se representar relacionamentos entre objetos e entre classes, respectivamente. Uma ligação é uma conexão entre objetos. No exemplo anterior, há uma ligação entre os objetos `João` e `Pessoal`. Uma associação, por sua vez, descreve um conjunto de ligações com estrutura e semântica comuns. No exemplo anterior, há uma associação entre as classes `empregado` e `departamento`. Todas as ligações de uma associação interligam objetos das mesmas classes, e assim, uma associação descreve um conjunto de potenciais ligações da mesma maneira que uma classe descreve um conjunto de potenciais objetos [Rumbaugh94].

Composição ou Agregação

Uma forma especial de relacionamento entre objetos é a *composição* ou *agregação*. Parte do poder dos softwares orientados a objetos advém de sua habilidade de manipular objetos complexos, como sendo compostos de vários outros objetos mais simples. Um carro, por exemplo, é composto por motor, rodas, carroceria, etc... Um motor, por sua vez, é composto de bloco, válvulas, pistões, e assim por diante.

Composição é o relacionamento *todo-parte/uma-parte-de* onde os objetos representando os componentes de alguma coisa são associados a um objeto representando o todo. Em outras palavras, a composição é um tipo forte de associação, onde um objeto agregado é composto de vários objetos componentes [Rumbaugh94].

Generalização / Especialização

Muitas vezes, um conceito geral pode ser especializado, adicionando-se novas características. Tomemos, como exemplo, o conceito que temos de *estudantes*. De modo geral, há características que são intrínsecas a quaisquer estudantes. Entretanto, é possível especializar este conceito para mostrar especificidades de subtipos de estudantes, tais como estudantes de 1º grau, estudantes de 2º grau, estudantes de graduação e estudantes de pós-graduação, entre outros.

Da maneira inversa, pode-se extrair de um conjunto de conceitos, características comuns que, quando generalizadas, formam um conceito geral. Por exemplo, ao avaliarmos

os conceitos que temos de carros, motos, caminhões e ônibus, podemos notar que esses têm características comuns que podem ser generalizadas em um supertipo *veículos automotores terrestres*.

As abstrações de especialização e generalização são muito úteis para a estruturação de sistemas. Com elas, é possível construir hierarquias de classes, subclasses, subsubclasses, e assim por diante.

A *herança* é um mecanismo para modelar similaridades entre classes, representando as abstrações de generalização e especialização. Através da herança, é possível tornar explícitos atributos e serviços comuns em uma hierarquia de classes. O mecanismo de herança possibilita reutilização, captura explícita de características comuns e definição incremental de classes.

Freqüentemente, promove-se a herança como a idéia central para a reutilização na indústria de software. Entretanto, apesar da herança, quando adequadamente usada, ser um mecanismo muito útil em diversos contextos, incluindo reuso, ela não é um pré-requisito para a reutilização.

Uma das principais vantagens da herança é facilitar a modificação de modelos. A herança nos permite conceber uma nova classe como um refinamento de outras classes. A nova classe pode herdar as similaridades e definir apenas a funcionalidade nova. O desenvolvimento orientado a objetos é fortemente baseado na identificação de objetos e na construção de *hierarquias de classes*, utilizando o mecanismo de herança.

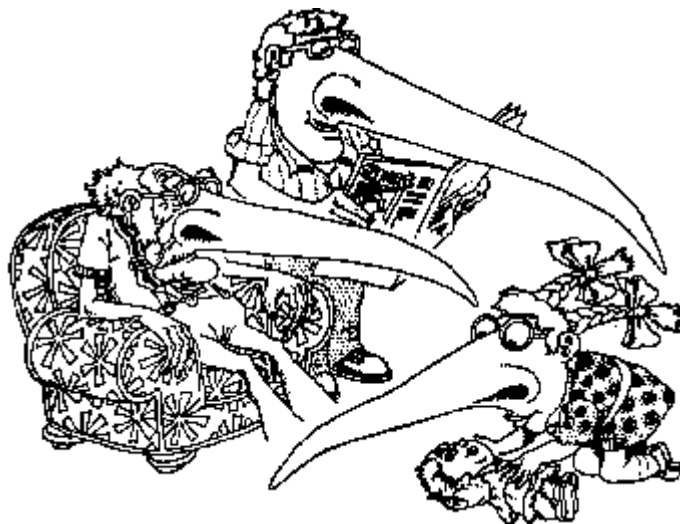


Figura 4.5 - Uma subclasse herda estrutura e comportamento de suas superclasses [Booch94].

Muitas vezes alguns objetos não se enquadram satisfatoriamente às propriedades descritas em uma classe, tipicamente porque:

- Além das propriedades descritas na classe, esses objetos possuem outras ainda não descritas;
- Algumas das propriedades descritas para a classe não são adequadas aos novos objetos, sendo necessário, portanto, redefini-las ou mesmo cancelá-las. Vale ressaltar que o cancelamento de propriedades, contudo, é um indicador de que a hierarquia não está modelada adequadamente.

Através do mecanismo de herança, tais problemas podem ser contornados. A herança define o relacionamento entre classes, no qual uma classe compartilha a estrutura e comportamento definido em uma ou mais outras classes. A classe que herda características² é chamada *subclasse* e a que fornece as características, *superclasse*. Desta forma, a herança representa uma hierarquia de abstrações na qual uma subclasse herda de uma ou mais superclasses.

Tipicamente, uma subclasse aumenta ou redefine características das superclasses. Assim, se uma classe B herda de uma classe A, todas as características descritas em A tornam-se automaticamente parte de B, que ainda é livre para acrescentar novas características para seus propósitos específicos.

A generalização permite abstrair, a partir de um conjunto de classes, uma classe mais geral contendo todas as características comuns. A especialização é a operação inversa e portanto, permite especializar uma classe em um número de subclasses, explicitando as diferenças entre as novas subclasses. Deste modo é possível compor a hierarquia de classes. Esses tipos de relacionamento são conhecidos também como relacionamentos “*é um tipo de*”, onde um objeto da subclasse também “*é um tipo de*” objeto da superclasse. Neste caso uma instância da subclasse é dita uma *instância indireta* da superclasse.

Quando uma subclasse herda características de uma única superclasse, tem-se *herança simples*. Quando uma classe é definida a partir de duas ou mais superclasses, tem-se *herança múltipla*. É importante observar, no entanto, que na herança múltipla podem ocorrer dois problemas: colisão de nomes herdados a partir de diferentes superclasses e a possibilidade de herança repetida. A figura 4.6 ilustra estes dois casos.

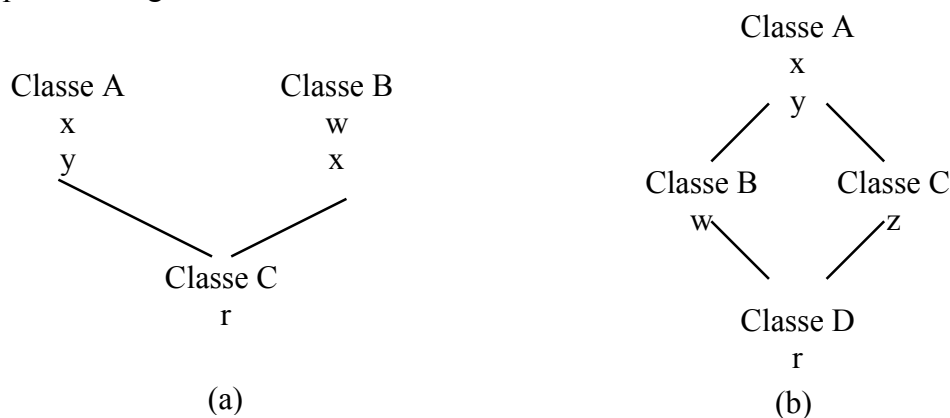


Figura 4.6 - (a) Colisão de nomes. (b) Herança repetida.

No primeiro caso, a classe C herda das classes A e B. Entretanto ambas possuem uma característica com nome x. Assim, como será a característica x em C, igual à definida na classe A ou igual à da classe B?

No segundo caso, a classe D herda das classes B e C, que por sua vez herdam da classe A. Assim, temos um caso de herança repetida, já que, indiretamente, a classe D herda duas vezes da classe A.

O resultado líquido da herança é que desenvolvedores podem evitar a codificação de redundâncias através da localização de cada operação no nível apropriado na hierarquia de classes.

² Usaremos o termo *característica* para designar estrutura (atributos) e comportamento (operações).

Mensagens e Métodos

A abstração incorporada por um objeto é caracterizada por um conjunto de operações que podem ser requisitadas por outros objetos, ditos clientes. Métodos são implementações reais de operações. Para que um objeto realize alguma tarefa, é necessário enviar a ele uma mensagem, solicitando a execução de um método específico. Um cliente só pode acessar um objeto através da emissão de mensagens, isto é, ele não pode acessar ou manipular diretamente os dados associados ao objeto. Os objetos podem ser complexos e o cliente não precisa tomar conhecimento de sua complexidade interna. O cliente precisa saber apenas como se comunicar com o objeto e como ele reage.

As mensagens são o meio de comunicação entre objetos e são responsáveis pela ativação de todo e qualquer processamento. Dessa forma, é possível garantir que clientes não serão afetados por alterações nas implementações de um objeto que não alterem o comportamento esperado de seus serviços.

4.2.3 - Conceitos Avançados

Classes e Operações Abstratas

Nem todas as classes são projetadas para instanciar objetos. Algumas são usadas simplesmente para organizar características comuns a diversas classes ou para encapsular classes que participam de uma mesma associação ou composição. Tais classes são ditas classes *classes abstratas*. Uma classe abstrata é desenvolvida basicamente para ser herdada por outras classes. Ela existe meramente para que um comportamento comum a um conjunto de classes possa ser fatorado em uma localização comum e definido uma única vez. Assim, uma classe abstrata não possui instâncias diretas mas suas classes descendentes *concretas*, sim. Uma classe concreta é uma classe instanciável, isto é, que pode ter instâncias diretas. Uma classe abstrata pode ter subclasses também abstratas, mas as classes-folhas na árvore de herança devem ser classes concretas.

Classes abstratas podem ser projetadas de duas maneiras distintas. Primeiro, elas podem prover implementações completamente funcionais do comportamento que pretendem capturar. Alternativamente, elas podem prover apenas definição de um protocolo para uma operação sem apresentar um método correspondente. Tal operação é dita uma *operação genérica ou abstrata*. Neste caso, a classe abstrata não é completamente implementada e todas as suas subclasses concretas são obrigadas a prover uma implementação para suas operações abstratas. Assim, diz-se que uma operação abstrata é uma operação com múltiplas implementações. Ela define apenas a assinatura³ a ser usada nas implementações que as subclasses deverão prover, garantindo, assim, uma interface consistente. Métodos que implementam uma operação genérica têm a mesma semântica. Uma vez que a mesma operação é definida em várias classes de uma mesma hierarquia, é importante que os métodos que a implementam conservem sua interface com o exterior (assinatura), isto é, o nome, o número e o tipo dos argumentos, e os resultados da operação.

³ nome da operação, parâmetros e retorno

Uma classe concreta não pode conter operações abstratas porque senão seus objetos teriam operações indefinidas. Analogamente, toda classe que possuir uma operação genérica não pode ter instâncias diretas e, portanto, obrigatoriamente é uma classe abstrata.

Sobrecarga

Em sistemas orientados a objetos, operações distintas de classes distintas, ou até de uma mesma classe, podem ter o mesmo nome. Neste caso, temos um nome de operação sobrecarregado.

Polimorfismo

O polimorfismo é uma poderosa ferramenta para o desenvolvimento de sistemas flexíveis. Polimorfismo significa a habilidade de tomar várias formas. No contexto da orientação a objetos, o polimorfismo está intrinsecamente ligado à comunicação entre objetos. De fato, polimorfismo pode ser melhor caracterizado, neste contexto, como o fato de um objeto emissor de uma mensagem não precisar conhecer a classe do objeto receptor. Assim, uma mensagem pode ser interpretada de diferentes maneiras, dependendo da classe do objeto receptor, ou seja, é o objeto que receptor que determina a interpretação da mensagem, e não o objeto emissor. O emissor precisa saber apenas que o receptor pode realizar certo comportamento, mas não a que classe ele pertence e, portanto, que operação é efetivamente executada. Um objeto sabe qual é a sua classe, e, portanto, a correta implementação da operação requisitada. A mensagem é associada ao método a ser realmente executado, através da identificação da operação e da classe do objeto receptor.

Freqüentemente, o polimorfismo é caracterizado como o fato de uma operação poder ser implementada de diferentes maneiras em diferentes classes. Todavia, isto é apenas uma consequência do que foi dito anteriormente e não polimorfismo em si.

A maioria dos autores não diferencia sobrecarga e polimorfismo, tratando ambos os casos como polimorfismo. Neste texto, entretanto, preferimos utilizar polimorfismo com uma semântica mais rígida: polimorfismo limitado a uma hierarquia de classes. Uma operação será dita polimórfica se ela existir, com a mesma assinatura, em uma cadeia de superclasses-subclasses. Uma operação polimórfica tem uma única semântica e os métodos que a implementam conservam essa semântica e, por conseguinte, a sua interface (assinatura). Na sobrecarga de operador, as operações não têm necessariamente a mesma semântica e não há necessidade de se preservar a assinatura. Ao contrário, se as operações sobrecarregadas forem da mesma classe, elas deverão ter assinaturas diferentes. Na prática as operações sobrecarregadas são, normalmente, uma coincidência na escolha de nomes de operação.

De um outro ponto de vista, o polimorfismo pode ser considerado uma característica dos objetos, refletindo a capacidade deles mudarem de classe em tempo de execução. O polimorfismo é extremamente útil quando deseja-se executar alguma operação em um objeto, mas não se sabe, a priori, qual será exatamente a sua classe em tempo de execução.

Ligação Dinâmica

Quando uma mensagem é enviada a um objeto, deve ser estabelecida uma ligação entre a mensagem enviada e o método a ser executado. A seleção do código para executar uma operação, como dito anteriormente, é baseada nos objetos identificados nas mensagens.

Quando a classe do objeto que recebe a mensagem é conhecida em tempo de compilação, a ligação pode ser feita nesta etapa ou durante a link-edição. Neste caso, tem-se

ligação estática. No entanto, muitas vezes, a classe de um objeto só pode ser identificada quando a mensagem é realmente emitida e, portanto, o código só pode ser selecionado neste momento. Neste caso, tem-se *ligação dinâmica* ou *tardia* (*late binding*).

Muitas vezes, polimorfismo e ligação dinâmica são confundidos. Porém, é importante frisar que ligação dinâmica significa apenas que a mensagem só é associada ao método específico da classe do objeto receptor no momento em que é enviada.

4.3 – O Processo de Desenvolvimento Orientado a Objetos

No desenvolvimento de grandes sistemas, uma abordagem sistemática deve ser adotada. Várias abordagens têm sido propostas, todas objetivando a produção de bons sistemas. Mas o que é um bom sistema? Segundo Jacobson [Jacobson92], para responder a essa pergunta, dois pontos de vista devem ser observados:

- Do ponto de vista externo, isto é, dos usuários, um bom sistema deve ser correto, rápido, confiável, fácil de ser usado, eficiente, etc...
- Do ponto de vista interno, ou seja, dos desenvolvedores, um bom sistema deve ser fácil de ser entendido e modificado, reutilizável, compatível com outros sistemas, portátil, etc...

Além disso, a definição de um bom sistema varia, geralmente, em função de sua aplicação.

Apesar de haver muitas abordagens documentadas para análise e projeto orientados a objetos, há muito pouca informação disponível sobre processos de desenvolvimento orientados a objetos. Um processo de desenvolvimento engloba um conjunto de atividades, métodos, técnicas e práticas que guiam as pessoas na produção de software, permitindo que um produto seja coerentemente criado. Um processo eficaz deve, claramente, considerar as relações entre as atividades, os artefatos requeridos e produzidos, os recursos, ferramentas e procedimentos necessários e a habilidade, o treinamento e a motivação do pessoal envolvido.

Processos de desenvolvimento não são sempre necessários. Em pequenos projetos, desenvolvedores podem se comunicar informalmente, dado o pequeno número de pessoas envolvidas. À medida que o número de desenvolvedores cresce, contudo, os canais de comunicação informais não são mais confiáveis e um processo de desenvolvimento é, então, necessário. De fato, nestes casos, a definição de um processo de desenvolvimento é um elemento essencial para assegurar a qualidade em um projeto.

Há vários aspectos a serem considerados na definição de um processo de software. No centro de sua arquitetura estão as atividades-chave do processo: planejamento, levantamento de requisitos, análise, projeto, implementação e testes, que são a base sobre a qual o processo de desenvolvimento deve ser construído. Entretanto, um processo envolve a escolha de um modelo de ciclo de vida, o detalhamento de suas macro-atividades, a escolha de métodos e técnicas para a sua realização e a definição de recursos e artefatos necessários e produzidos.

Um processo de desenvolvimento de software não pode ser definido de forma universal. Para ser eficaz e conduzir à construção de produtos de boa qualidade, um processo deve ser adequado ao domínio da aplicação e ao projeto específico. Deste modo, processos devem ser definidos caso a caso, considerando-se as especificidades da aplicação, a tecnologia a ser adotada na sua construção, a organização onde o produto será desenvolvido e o grupo de desenvolvimento.

Em suma, o objetivo de se definir um processo de software é favorecer a produção de sistemas de alta qualidade, atingindo as necessidades dos usuários finais, dentro de um cronograma e um orçamento previsíveis.

A escolha de um modelo de ciclo de vida é o ponto de partida para a definição de um processo de software. Um modelo de ciclo de vida organiza as macro-atividades básicas, estabelecendo precedência e dependência entre as mesmas.

Um ciclo de vida pode ser entendido como passos ou atividades que devem ser executados durante um projeto. Para a definição completa do processo, a cada atividade, devem ser associados técnicas, ferramentas e critérios de qualidade, entre outros, formando uma base sólida para o desenvolvimento. Adicionalmente, outras atividades tipicamente de cunho gerencial, devem ser definidas, entre elas gerência de configuração e controle e garantia da qualidade.

De maneira geral, o ciclo de vida de um software envolve as seguintes fases:

- *Planejamento*: O objetivo do planejamento de projeto é fornecer uma estrutura que possibilite ao gerente fazer estimativas razoáveis de recursos, custos e prazos. Uma vez estabelecido o escopo de software, uma proposta de desenvolvimento deve ser elaborada, isto é, um plano de projeto deve ser elaborado configurando o processo a ser utilizado no desenvolvimento de software. À medida que o projeto progride, o planejamento deve ser detalhado e atualizado regularmente. Pelo menos ao final de cada uma das fases do desenvolvimento (levantamento de requisitos, análise, projeto, implementação e teste) o planejamento como um todo deve ser revisto e o planejamento da etapa seguinte deve ser detalhado.
- *Levantamento de Requisitos*: Nesta fase, o processo de coleta (levantamento) de requisitos é intensificado. O escopo deve ser refinado e os requisitos identificados. Para entender a natureza do software a ser construído, o engenheiro de software tem de compreender o domínio do problema, bem como a funcionalidade e o comportamento esperados.
- *Análise*: Uma vez identificados os requisitos do sistema a ser desenvolvido, estes devem ser modelados, avaliados e documentados. Uma parte vital desta fase é a construção de um modelo descrevendo *o que* o software tem de fazer (e não *como* fazê-lo).
- *Projeto*: Esta fase é responsável por incorporar requisitos tecnológicos aos requisitos essenciais do sistema, modelados na fase anterior e, portanto, requer que a plataforma de implementação seja conhecida. Basicamente, envolve duas grandes etapas: projeto da arquitetura do sistema e projeto detalhado. O objetivo da primeira etapa é definir a arquitetura geral do software, tendo por base o modelo construído na fase de análise de requisitos. Esta arquitetura deve descrever a estrutura de nível mais alto da aplicação e identificar seus principais componentes. O propósito do projeto detalhado é detalhar o projeto do software para cada componente identificado na etapa anterior. Os componentes de software devem ser sucessivamente refinados em níveis de maior detalhamento, até que possam ser codificados e testados.
- *Implementação*: O projeto deve ser traduzido para uma forma passível de execução pela máquina. A fase de implementação realiza esta tarefa, isto é, cada unidade de software do projeto detalhado é implementada.

- *Testes*: inclui diversos níveis de testes, a saber, teste de unidade, teste de integração e teste de sistema. Inicialmente, cada unidade de software implementada deve ser testada e os resultados documentados. A seguir, os diversos componentes devem ser integrados sucessivamente até se obter o sistema. Finalmente, o sistema como um todo deve ser testado.
- *Implantação*: uma vez testado, o software deve ser colocado em produção. Para tal, contudo, é necessário treinar os usuários, configurar o ambiente de produção e, muitas vezes, converter bases de dados. O propósito desta fase é estabelecer que o software satisfaz os requisitos dos usuários. Isto é feito instalando o software e conduzindo testes de aceitação (validação). Quando o software tiver demonstrado prover as capacidades requeridas, ele pode ser aceito e a operação iniciada.
- *Operação*: nesta fase, o software é utilizado pelos usuários no ambiente de produção.
- *Manutenção*: Indubitavelmente, o software sofrerá mudanças após ter sido entregue para o usuário. Alterações ocorrerão porque erros foram encontrados, porque o software precisa ser adaptado para acomodar mudanças em seu ambiente externo, ou porque o cliente necessita de funcionalidade adicional ou aumento de desempenho. Muitas vezes, dependendo do tipo e porte da manutenção necessária, esta fase pode requerer a definição de um novo processo, onde cada uma das fases precedentes é re-aplicada no contexto de um software existente ao invés de um novo.

Uma vez que o software é sempre parte de um sistema (ou negócio) maior, o trabalho começa pelo estabelecimento dos requisitos para todos os elementos do sistema e, na sequência, procede-se a alocação para software de algum subconjunto destes requisitos. Esta etapa é a Engenharia de Sistemas e antecede a todas as demais relacionadas.

Um *modelo de ciclo de vida* estrutura as atividades do projeto em *fases* e define como estas fases estão relacionadas. A escolha de um modelo de ciclo de vida é fortemente dependente das características do projeto. Assim, é importante apresentar vários modelos de ciclo de vida adequados ao desenvolvimento orientado a objetos, indicando em que situações são aplicáveis. Dentre os principais modelos de ciclo de vida, destacam-se o modelo seqüencial linear, o modelo incremental e vários modelos evolutivos.

4.3.1 - O Modelo Seqüencial Linear

Algumas vezes chamado de “ciclo de vida clássico” ou “modelo em cascata”, o modelo seqüencial linear sugere uma abordagem seqüencial sistemática para o desenvolvimento de software que começa no nível de sistema e avança através de análise, projeto, implementação, teste e manutenção, como mostra a figura 4.7.

Na abordagem em cascata, as fases são executadas seqüencialmente. Cada fase é executada uma vez, embora um retorno à fase anterior seja permitido para correção de erros. A entrega do sistema completo ocorre em um único marco, ao final da fase de Testes de Validação e Implantação.

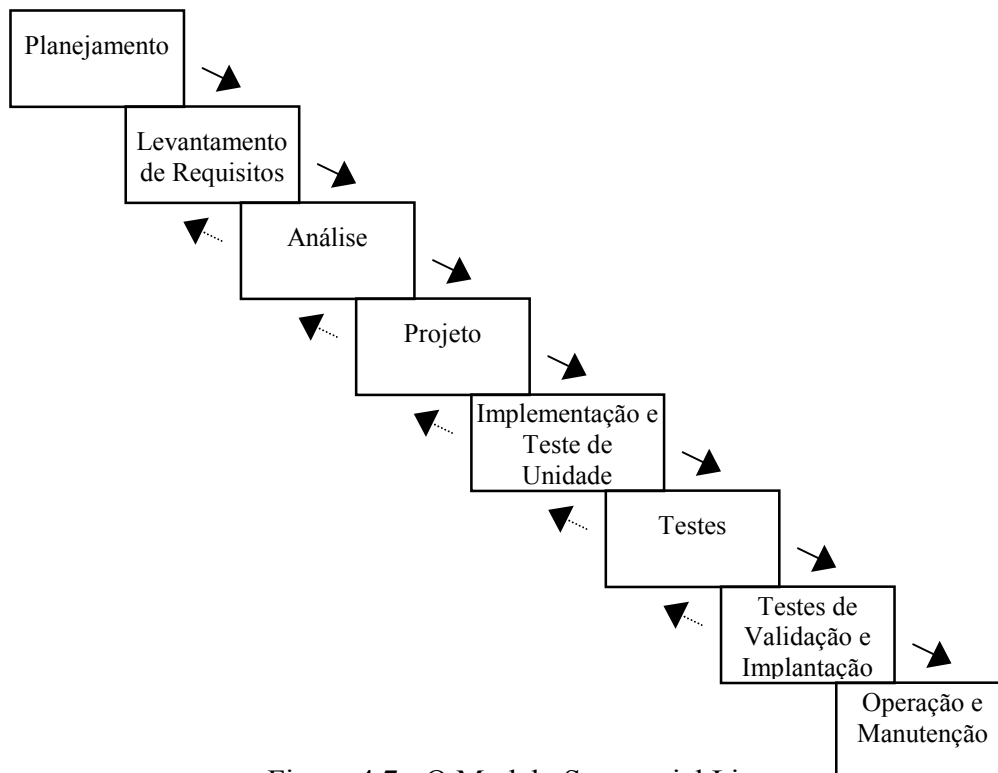


Figura 4.7 - O Modelo Sequencial Linear.

O modelo sequencial linear é o modelo de ciclo de vida mais antigo e mais amplamente usado. Entretanto, críticas têm levado ao questionamento de sua eficiência. Dentre os problemas algumas vezes encontrados na sua aplicação, destacam-se:

- Projetos reais muitas vezes não seguem o fluxo sequencial que o modelo propõe.
- Frequentemente, é difícil para o usuário colocar todos os requisitos explicitamente. O modelo sequencial linear requer isto e tem dificuldade de acomodar a incerteza natural que existe no início de muitos projetos.
- O usuário precisa ser paciente. Uma versão operacional do software não estará disponível até o final do projeto.
- A introdução de certos membros da equipe, tais como projetistas e programadores, é frequentemente adiada desnecessariamente. A natureza linear do ciclo de vida clássico leva a “estados de bloqueio” nos quais alguns membros da equipe do projeto precisam esperar que outros membros da equipe completem tarefas dependentes.

Cada um desses problemas é real. Entretanto, o modelo de ciclo de vida clássico tem um lugar definitivo e importante no trabalho de engenharia de software. Embora tenha fraquezas, ele é significativamente melhor do que uma abordagem casual para o desenvolvimento de software. De fato, para problemas pequenos e bem-definidos, onde os desenvolvedores conhecem bem o domínio do problema e os requisitos podem ser estabelecidos claramente, este modelo deve ser o preferido, uma vez que é o mais fácil de ser gerenciado.

4.3.2 - O Modelo Incremental

O modelo incremental é uma variação do modelo de ciclo de vida sequencial linear, na qual as fases de levantamento de requisitos, análise e projeto da arquitetura são realizadas para o software como um todo. Uma vez definida a arquitetura do software, as demais fases (projeto detalhado, implementação e teste) são divididas em unidades mais gerenciáveis e, assim sendo, o software é distribuído em várias versões, cada uma delas com funcionalidade e capacidade aumentadas, como mostra a figura 4.8.

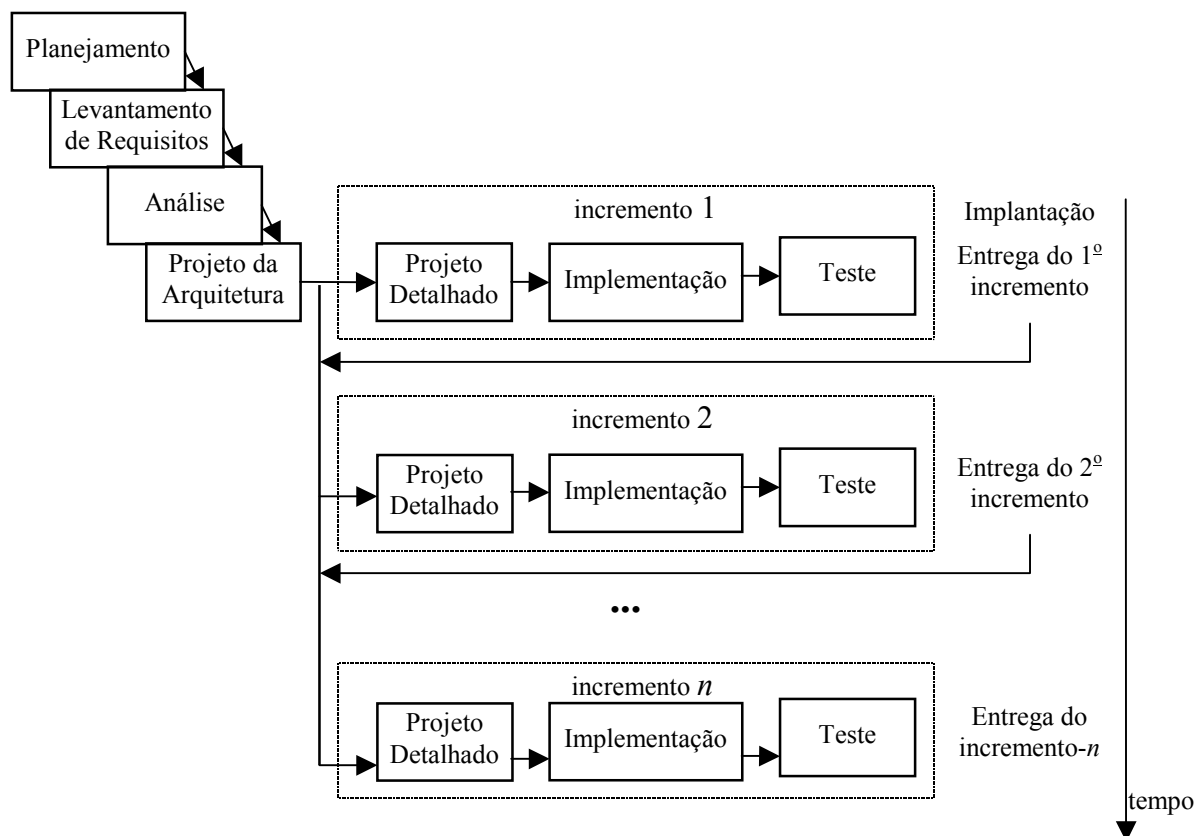


Figura 4.8 - O Modelo Incremental.

Este modelo de processo aplica-se quando o software a ser desenvolvido é grande, mas o problema é bem-definido, e não há recursos para seu desenvolvimento como um todo, ou quando se deseja apresentar resultados para o cliente mais rapidamente. De fato, ao se adotar este modelo, está-se estabelecendo um critério para o escalonamento do trabalho de projeto detalhado, implementação e testes.

4.3.3 - Modelos Evolutivos

Há um crescente reconhecimento que software, assim como todo sistema complexo, evolui por um período de tempo. Requisitos do negócio e do produto freqüentemente mudam à medida que o desenvolvimento avança, abrindo caminho para um produto final irreal; prazos apertados tornam impossível o término de um produto de software completo, mas uma versão limitada tem de ser introduzida para satisfazer às pressões do negócio ou à competitividade; um conjunto de requisitos centrais do produto ou sistema é bem

compreendido, mas os detalhes de extensões da aplicação têm ainda de ser definidos. Nestas e em situações similares, engenheiros de software necessitam de um modelo de ciclo de vida que tenha sido explicitamente projetado para acomodar um produto que evolua ao longo do tempo. Assim, quando o problema não é bem definido e ele não pode ser totalmente especificado no início do desenvolvimento, deve-se optar por um modelo evolutivo.

Os modelos evolutivos aplicam seqüências lineares de modo escalonado, à medida que o calendário de tempo avança. Cada seqüência linear produz um “incremento” distribuível do software que é colocado em uso. Durante o uso, novos requisitos são levantados, dando início a um novo ciclo de desenvolvimento. Desta forma, estes modelos permitem que os engenheiros de software desenvolvam iterativamente versões do software cada vez mais completas.

Modelo Espiral

O *modelo espiral* é capaz de descrever como um produto se desenvolve para formar novas versões e como uma versão pode ser incrementalmente desenvolvida, partindo-se de um protótipo até um produto completo. A figura 4.9 ilustra este modelo.

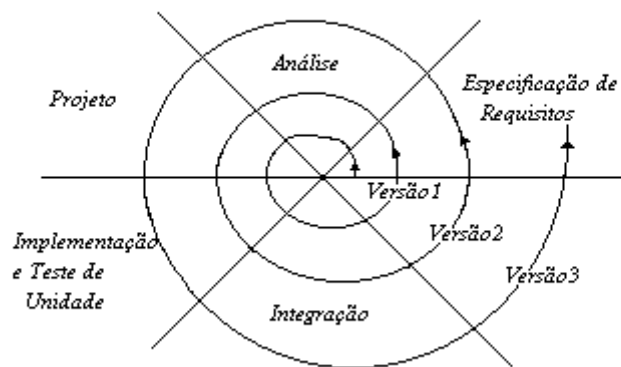


Figura 4.9 - Um modelo espiral [Jacobson92].

Uma das importantes características do desenvolvimento de software orientado a objetos é que *todas* as atividades do ciclo de vida compartilham vocabulário, notação e estratégias comuns, isto é, tudo gira em torno de objetos. Seja na fase de análise, projeto ou implementação, a equipe de desenvolvimento estará sempre envolvida na descoberta, documentação, prototipação e/ou desenvolvimento de objetos, e em *todas* as atividades do ciclo de vida, os conceitos de abstração fundamentais para a orientação a objetos, tais como encapsulamento e herança, desempenham um importante papel. Esta é uma das principais diferenças em relação às metodologias convencionais, onde, por exemplo, as atividades do grupo de modelagem de dados freqüentemente parecem não ter qualquer relação com as atividades do grupo de modelagem funcional, sendo que as notações usadas por cada um dos grupos são completamente díspares. Além disso, sistemas orientados a objetos apresentam outras características muito interessantes e desejáveis ao processo de produção de software, entre elas:

- visão do mundo real mais adequada através da observação de objetos, com conseqüente *redução do gap semântico*, que proporciona uma visão balanceada de dados e processos;
- *desenvolvimento incremental e evolutivo*. Esta característica é extremamente desejável para que um produto possa ser desenvolvido em etapas ou por equipes distintas;
- *reusabilidade*. Através da reutilização, muitas vezes é possível reaproveitar parcelas de código, projetos ou mesmo de especificações de requisitos na construção de outras partes de um sistema, ou até mesmo de outros sistemas;
- possibilidade de incorporação de pequenas diferenças a elementos do sistema, através da *abstração de generalização/especialização*, mantendo alta coesão do sistema;
- *modularidade*. O conceito de objetos e sua descrição em classes, incorporando dados e operações, constituem critérios de modularização altamente efetivos e propiciam o encapsulamento.

Dadas estas características, modelos em espiral e modelos baseados em prototipação têm sido amplamente utilizados no processo de desenvolvimento de sistemas orientados a objetos.

Modelo Evolutivo Básico

O modelo evolutivo básico pressupõe que o desenvolvimento se dará em ciclos, onde ao final de cada ciclo, uma *versão operacional do software* é colocada em uso. Durante o uso, novos requisitos são levantados, dando início a um novo ciclo no desenvolvimento. A figura 4.10 ilustra este modelo.

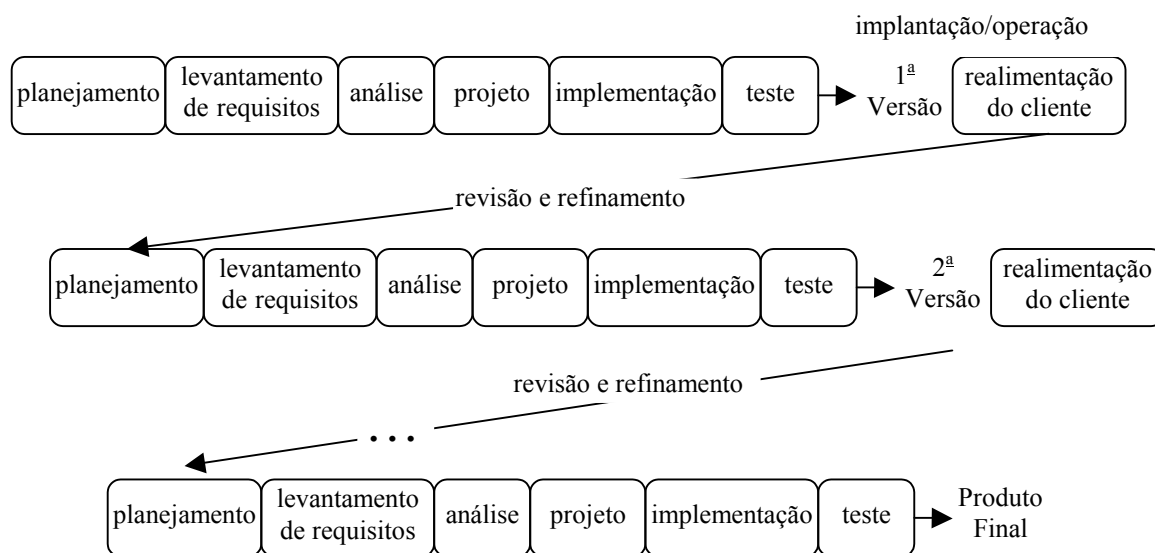


Figura 4.10 - O modelo evolutivo básico.

O primeiro incremento é geralmente um *produto central*, isto é, os requisitos básicos são tratados, mas muitas características suplementares (algumas conhecidas, outras não) permanecem indisponíveis. O produto central é usado pelo cliente (ou passa por uma revisão detalhada). Como resultado do uso e/ou da avaliação, um plano é desenvolvido para o próximo incremento. Este plano endereça dois aspectos principais: a modificação do produto

central para melhor satisfazer as necessidades do cliente e a entrega de funcionalidades e características adicionais. Este processo é repetido, seguindo a entrega de cada incremento, até que o produto completo seja construído.

O desenvolvimento evolutivo é particularmente útil quando não há recursos de pessoal disponíveis para uma completa implementação nos prazos estabelecidos para o projeto. Além disso, incrementos podem ser planejados para gerenciar riscos técnicos.

O Modelo Rational

A Rational Software desenvolveu um modelo de processo, integrado com uma coleção de ferramentas, para o desenvolvimento de sistemas complexos, orientados a objetos [Kruchten98], que pode ser adaptado e estendido para adequar-se a projetos específicos.

O ciclo de vida adotado por este processo é tipicamente evolutivo. Contudo, uma forma de organização em fases é adotada para comportar os ciclos de desenvolvimento, permitindo uma gerência mais efetiva no projeto de sistemas complexos. Ao contrário do tradicionalmente definido como fases na maioria dos modelos de ciclo de vida – planejamento, levantamento de requisitos, análise, projeto, implementação e testes – são definidas fases ortogonais a estas, a saber [Kruchten98]:

- **Concepção:** nesta fase, é estabelecido o escopo do projeto e suas fronteiras, discriminando os principais casos de uso do sistema. Deve-se estimar os custos e prazos globais para o projeto como um todo e prover estimativas detalhadas para a fase de elaboração. Ao término desta fase, são examinados os objetivos do projeto para se decidir sobre a continuidade do desenvolvimento;
- **Elaboração:** o propósito desta fase é analisar mais refinadamente o domínio do problema, estabelecer uma arquitetura de fundação sólida, desenvolver um plano de projeto para o sistema a ser construído e eliminar os elementos de projeto que oferecem maior risco. Ao término desta fase, a parte considerada mais complicada do desenvolvimento é considerada completa. Embora o processo deva sempre acomodar alterações, as atividades da fase de elaboração asseguram que os requisitos, a arquitetura e os planos estão suficientemente estáveis e que os riscos estão suficientemente mitigados, de modo a se poder prever com precisão os custos e prazos para a conclusão do desenvolvimento.
- **Construção:** durante a fase de construção, um produto completo é desenvolvido de maneira iterativa e incremental para que esteja pronto para a transição à comunidade usuária.
- **Transição:** nesta fase, o software é disponibilizado à comunidade usuária. Após o produto ter sido colocado em uso, naturalmente surgem novas considerações que irão demandar a construção de novas versões para permitir ajustes do sistema, corrigir problemas ou concluir algumas características que foram postergadas.

É importante realçar que dentro de cada fase, um conjunto de iterações, envolvendo planejamento, levantamento de requisitos, análise, projeto e implementação e testes, é realizado. Contudo, de uma iteração para outra e de uma fase para a próxima, a ênfase sobre as várias atividades muda, como mostra a figura 4.11. Na fase de concepção, o foco principal recai sobre o entendimento dos requisitos e a determinação do escopo do projeto (planejamento e levantamento de requisitos). Na fase de elaboração, o enfoque está na captura e modelagem dos requisitos (levantamento de requisitos e análise), ainda que algum trabalho

de projeto e implementação seja realizado para prototipar a arquitetura, evitando certos riscos técnicos. Na fase de construção, o enfoque concentra-se no projeto e na implementação, visando evoluir e recheiar o protótipo inicial, até obter o primeiro produto operacional. Finalmente, a fase de transição concentra-se na garantia de que o sistema possui o nível adequado de qualidade (testes). Além disso, usuários devem ser treinados, características ajustadas e elementos esquecidos adicionados.

	Levantamento de Requisitos	Análise	Projeto	Implementação	Testes
Concepção					
Elaboração					
Construção					
Transição					

Figura 4.11 – A ênfase principal de cada uma das fases.

Além dos conjuntos de iterações em cada fase, o conjunto de fases em si pode ser iterativo. Como mostra a figura 4.12, as fases não necessariamente têm a mesma duração. O esforço realizado em cada fase irá variar fortemente em função das circunstâncias específicas do projeto.

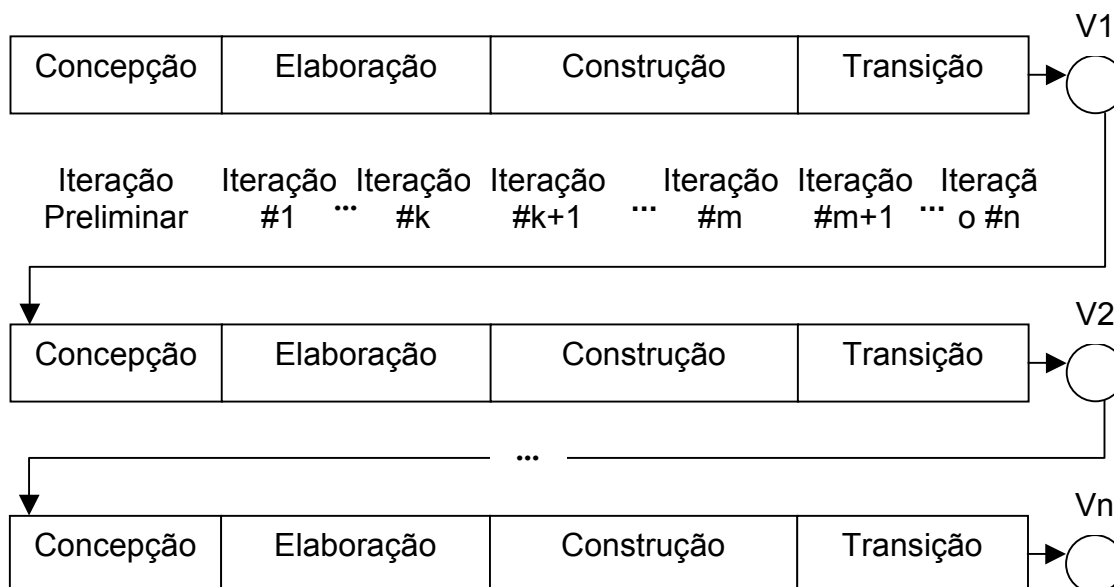


Figura 4.12 – O Modelo da Rational.

A figura 4.13 mostra uma distribuição de tempo típica para ciclos de desenvolvimento iniciais de um projeto [Kruchten98].

Concepção 10%	Elaboração 30%	Construção 50%	Transição 10%
------------------	-------------------	-------------------	------------------

Figura 4.13 – Uma distribuição de tempo típica para ciclos iniciais.

Durante a captura e descrição detalhada dos casos de uso, o sistema é cuidadosamente estudado. Uma vez que, geralmente, casos de uso enfocam uma particular funcionalidade, é possível analisar a funcionalidade total do sistema de maneira incremental. Deste modo, casos de uso para áreas funcionalmente diferentes podem ser desenvolvidos independentemente e, mais tarde, reunidos para formar um modelo de requisitos completo. Com isso, permite-se focar um problema de cada vez, abrindo caminho para um desenvolvimento incremental ou evolutivo.

4.4 - A Linguagem de Modelagem Unificada

A Linguagem de Modelagem Unificada (*Unified Modeling Language* – UML) é a linguagem padrão para especificar, visualizar, documentar e construir artefatos de um sistema e pode ser utilizada em todos os processos de desenvolvimento orientados a objetos, ao longo de seus ciclos de desenvolvimento, usando diferentes tecnologias de implementação [Furlan98].

A UML teve origem em uma tentativa de se unificar os principais métodos orientados a objetos utilizados até então: a OMT [Rumbaugh94] e o Método de Booch [Booch94]. A este esforço juntou-se também Ivar Jacobson, fundindo também seu método OOSE [Jacobson92]. Contudo, percebeu-se que não era possível estabelecer um único método adequado para todo e qualquer desenvolvimento. De fato, um método é composto por uma notação para os artefatos produzidos e de um processo descrevendo que artefatos construir e como construí-los. A notação pode ser unificada, mas a decisão de quais artefatos produzir e que passos seguir não é passível de padronização, já que varia de projeto para projeto. Assim, ao invés de criarem um método unificado, Rumbaugh, Booch e Jacobson propuseram a UML, incorporando as principais notações para os produtos de seus métodos e de vários outros, com a colaboração de várias empresas e autores. A UML foi aprovada em novembro de 1997 pela OMG – *Object Management Group* – pondo fim a uma guerra de métodos OO.

A UML pode ser usada para [Furlan98]:

- Mostrar as fronteiras de um sistema e suas funções principais, utilizando atores e casos de uso;
- Representar a estrutura estática de um sistema, utilizando diagramas de classes;
- Modelar o comportamento de objetos com diagramas de estados;
- Ilustrar a realização de casos de uso com diagramas de interação;
- Revelar a arquitetura de implementação física com diagramas de implementação.

A UML vai além de uma simples padronização em busca de uma notação unificada, uma vez que contém conceitos novos que não são encontrados em outros métodos orientados a objetos, como é o caso dos diagramas de atividade.

Os diagramas contemplados pela UML são [Furlan98]:

- Diagrama de Casos de Uso: Os casos de uso descrevem a funcionalidade do sistema percebida por atores externos. Um ator interage com o sistema, podendo ser um usuário, dispositivo ou outro sistema.
- Diagrama de Classes: Denota a estrutura estática de um sistema, isto é, as classes, os relacionamentos entre suas instâncias (objetos), restrições e hierarquias. O diagrama é considerado estático pois a estrutura descrita é sempre válida em qualquer ponto no ciclo de vida do sistema.
- Diagramas de Interação, podendo ser:
 - ✓ Diagramas de Seqüência: mostram a colaboração dinâmica entre um número de objetos, sendo seu objetivo principal mostrar a seqüência de mensagens enviadas entre objetos. É um gráfico bidimensional, onde a dimensão vertical representa o tempo e a dimensão horizontal os diferentes objetos.
 - ✓ Diagramas de Colaboração: têm exatamente o mesmo propósito dos diagramas de seqüência, apresentando, contudo, um formato diferente. São desenhados como diagramas de objetos, onde são mostradas as mensagens trocadas entre os objetos.
- Diagramas de Estados: mostram as seqüências de estados pelos quais um objeto pode passar ao longo de sua vida, em resposta a estímulos recebidos, juntamente com suas respostas e ações. É tipicamente um complemento de uma classe e relaciona os possíveis estados que os objetos da classe podem ter e quais eventos podem causar uma transição de um estado para outro. A UML descreve, também, uma variação dos Diagramas de Estado, na qual a maioria dos estados é estado de ação e a maioria das transições é ativada por conclusões das ações, os ditos Diagramas de Atividades;
- Diagramas de Implementação, composto por dois diagramas:
 - ✓ Diagrama de Componentes: são mostradas as dependências entre componentes de software, inclusive código fonte, código binário e componentes executáveis.
 - ✓ Diagrama de Implantação: mostra elementos de configuração do processamento em tempo de execução, isto é, os componentes de software, processos e dispositivos físicos.

Vale ressaltar mais uma vez, que a UML é uma linguagem de modelagem, não um método de desenvolvimento OO. Os métodos consistem, pelo menos em princípio, de uma linguagem de modelagem e um procedimento de uso dessa linguagem. A UML não prescreve explicitamente esse procedimento de utilização [Furlan98]. Assim, a UML deve ser aplicada no contexto de um processo, lembrando que domínios de problemas e projetos diferentes podem requerer processos diferentes.

Referências Bibliográficas do Capítulo

- [Booch94] G. Booch; *Object-Oriented Analysis and Design with Applications*, 2nd edition, Benjamin/Cummings Publishing Company, Inc, 1994.
- [Coad92] P. Coad, E. Yourdon; *Análise Baseada em Objetos*, Editora Campus, 1992.
- [Furlan98] J.D. Furlan; *Modelagem de Objetos Através da UML*; Makron Books, 1998.
- [Jacobson92] I. Jacobson; *Object-Oriented Software Engineering*, Addison-Wesley, 1992.
- [Kruchten98] P. Kruchten; *The Rational Unified Process: An Introduction*, Object Technology Series, Addison-Wesley, 1998.
- [Pompilho95] S. Pompilho. *Análise Essencial: Guia Prático de Análise de Sistemas*. IBPI Press, Editora Infobook, Rio de Janeiro, 1995.
- [Rumbaugh94] J. Rumbaugh, et alli; *Modelagem e Projetos Baseados em Objetos*, Editora Campus, 1994.
- [Snyder93] A. Snyder; “The Essence of Objects: Concepts and Terms”, IEEE Software, Janeiro 1993.
- [Yourdon94] E. Yourdon; *Object-Oriented Systems Design: an Integrated Approach*, Yourdon Press Computing Series, Prentice Hall, 1994.

5. Análise Orientada a Objetos

Quando um novo sistema precisa ser construído e decidimos utilizar o paradigma orientado a objetos (OO) em seu desenvolvimento, surge uma importante questão: Como modelar os requisitos do sistema de um modo adequado para a Engenharia de Software OO? Uma vez que, essencialmente, este paradigma trabalha com objetos, é necessário identificar quais os objetos relevantes, como eles se relacionam e como eles se comportam no contexto do sistema. Além disso, é preciso especificar e modelar o problema de maneira que seja possível criar um projeto OO efetivo. Todos estes aspectos são tratados dentro do contexto da Análise Orientada a Objetos (AOO) [Pressman00].

O propósito da Análise Orientada a Objetos (AOO) é definir todas as classes (e os relacionamentos e comportamentos associados a ela) que são relevantes para o problema a ser resolvido. Para tal, um número de tarefas deve ocorrer:

- Identificação de Classes
- Especificação de Hierarquias de Generalização/Especialização
- Identificação de Associações e Atributos
- Modelagem do Comportamento
- Definição das Operações

É importante notar que estas atividades são dependentes umas das outras e que, durante o desenvolvimento, elas são realizadas, tipicamente, de forma iterativa. A ênfase da fase de análise, no entanto, deve ser sempre o entendimento do domínio do problema, sempre desconsiderando aspectos de implementação.

A popularidade da orientação a objetos fez surgir dezenas de métodos OO para análise e projeto. Cada um deles, introduz um processo para analisar um sistema, um conjunto de modelos que evoluem ao longo do processo e uma notação que permite ao engenheiro de software criar cada modelo de maneira consistente [Pressman00]. Entre os principais métodos existentes podemos citar o Método de Booch [Booch94], OMT [Rumbaugh94], OOSE [Jacobson92] e o Método de Coad & Yourdon [Coad92][Coad93].

Ainda que, à primeira vista, estes métodos possam parecer substancialmente diferentes, isto não é verdade. Todos eles consideram, de alguma forma, as tarefas listadas anteriormente. Essencialmente, as maiores diferenças estão nas diretrizes e heurísticas fornecidas, nos modelos utilizados, suas notações e no processo sugerido. Tentativas para se definir um método padrão têm sido realizadas, mas têm encontrado como principal obstáculo a definição de um processo padrão de análise e projeto. Assim, um primeiro passo foi dado ao se propor a Linguagem de Modelagem Unificada (UML), que estabelece, para um conjunto de modelos normalmente utilizados no desenvolvimento OO, uma notação padrão.

Neste texto, não adotamos nenhum método específico, mas, ao mesmo tempo todos. Isto é, procuramos incorporar as características que julgamos serem as mais úteis de cada um dos métodos em uma abordagem básica para o desenvolvimento OO. O processo de desenvolvimento deve ser definido caso a caso, levando-se em conta as particularidades do projeto em questão, do domínio de aplicação e das características da equipe de desenvolvimento, entre outras. Mas a abordagem aqui proposta pode ser vista como o ponto de partida para esta definição. Quanto aos modelos e suas notações, adotamos, basicamente, aqueles definidos na UML [Furlan98].

5.1 - Identificação de Classes

Com base na especificação de requisitos e nos diagramas de casos de uso e suas descrições, é possível iniciar o trabalho de modelagem do sistema. O modelo de objetos dos requisitos dos usuários para o sistema a ser gerado deve ser tão independente de sua futura implementação quanto possível.

Não há nada mais central e crucial para qualquer método orientado a objetos do que o processo de descoberta de quais classes devem ser incluídas no modelo. O cerne de um modelo OO é exatamente o seu conjunto de classes [Yourdon94]. A figura 5.1 mostra a notação básica da UML para classes em um Diagrama de Classes.

Classe
Atributos da Classe
Operações da Classe

Figura 5.1 - Notação Básica da UML para Classes em um Diagrama de Classes.

Com a AOO, um engenheiro de software estuda, filtra e modela o domínio do problema. Dizemos que o engenheiro de software “filtra” o domínio, pois apenas uma parte desse domínio fará parte das responsabilidades do sistema. Assim, um domínio de problemas pode incluir várias informações e funcionalidades, mas as responsabilidades de um sistema neste domínio podem incluir apenas uma pequena parcela deste conjunto.

As classes de um modelo representam a expressão inicial do sistema. As atividades subsequentes da AOO buscam obter uma descrição cada vez mais detalhada do contexto, em termos de associações, atributos e operações. Assim, a identificação das classes é uma tarefa crucial para o desenvolvimento de um sistema OO. Para facilitar a identificação das classes do sistema [Coad92]:

- Estude o domínio de aplicação;
- Faça uma observação geral no ambiente real onde existe o problema;
- Procure ouvir atentamente os especialistas do domínio do problema;
- Verifique, se existirem, resultados de AOOs anteriores em domínios semelhantes;
- Observe outros sistemas no mesmo domínio ou em domínios similares;
- Consulte fontes bibliográficas;

Um aspecto fundamental no processo de análise é a interação constante com os especialistas de domínio. Técnicas de coleta de dados, tais como entrevistas, revisão de documentos e reuniões informais com os usuários, têm um papel importante nesta etapa. O espaço de problema em si, junto com quaisquer diagramas, figuras e informação textual providos pelos usuários, pode ser uma importante fonte para este trabalho. Para descobrir as classes de um domínio de aplicação, observe os seguintes elementos:

- objetos físicos ou conceituais que formam uma abstração coesa, sobre os quais o sistema precisa manipular informações;

- outros sistemas e “terminadores externos” que se comunicam ou interagem com o sistema em questão, atentando para as informações que estão sendo trocadas;
- dispositivos físicos com os quais o sistema em estudo terá de interagir, sem considerar a tecnologia para implementar o sistema em si;

Um primeiro passo no processo de identificação de classes deve ser a revisão e discussão da especificação de requisitos. Wirfs-Brock, Wilkerson, Wiener [Wirfs-Brock90] e Jacobson [Jacobson92] sugerem que uma boa estratégia para identificar objetos é ler a especificação de requisitos procurando por substantivos. Esses autores argumentam que um objeto é, tipicamente, descrito por um nome no domínio e, portanto, aprender sobre a terminologia do domínio do problema é um bom ponto de partida. Uma heurística menos vaga sugere que os seguintes elementos sejam considerados como potenciais candidatos a classes [Coad92]:

- coisas que são parte do domínio de informação do problema;
- ocorrências ou eventos que precisam ser registrados e lembrados pelo sistema;
- papéis desempenhados pelas diferentes pessoas que interagem direta ou indiretamente com o sistema;
- locais físicos ou geográficos e lugares que estabelecem o contexto do problema;
- unidades organizacionais (departamentos, divisões, etc...) que possam ser relevantes para o sistema.

Uma vez identificadas as potenciais classes, deve-se proceder uma avaliação para decidir o que considerar ou recusar. Dentre os critérios para inclusão de classes podem ser citados [Coad92]:

- Lembrança necessária: o sistema precisa se lembrar de alguma coisa sobre os objetos da classe? Normalmente, devem ser necessários, pelo menos, dois ou mais atributos;
- Funcionalidade necessária e essencial: deve ser possível identificar uma ou mais operações para as classe propostas, isto é, objetos destas classes têm de *fazer* algo para justificar a sua existência. Além disso, a funcionalidade identificada para a classe proposta deve ser relevante e necessária a despeito da tecnologia de hardware e software a ser usada para implementar o sistema; caso contrário, a classe proposta é uma classe de projeto ou de implementação e sua inclusão no modelo deve ser adiada até o respectivo estágio de desenvolvimento;
- Atributos e operações comuns: os atributos e as operações da classe devem ser aplicáveis a todas as suas instância, isto é, aos objetos da classe.

Para ser considerada uma classe legítima no modelo de análise, uma classe candidata tem de satisfazer a todas (ou quase todas) as características acima. Além disso, observe se existem várias instâncias da classe. Uma classe que possui uma única instância também não deve ser considerada uma classe.

O resultado principal desta atividade é a obtenção de uma lista de potenciais classes para o sistema em estudo.

5.2 - Especificação de Hierarquias de Generalização / Especialização

Um dos principais mecanismos de estruturação de conceitos é a generalização / especialização. Com este mecanismo é possível capturar similaridades entre classes, dispondo-as em hierarquias de classes.

A figura 5.2 mostra a notação da UML usada para representar a estrutura de generalização - especialização. É importante realçar que esta estrutura reflete um mapeamento entre classes e não entre objetos.

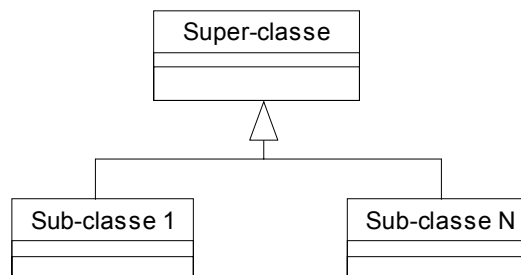


Figura 5.2 - Notação da UML para Hierarquia de Classes.

As seguintes diretrizes podem ser usadas analisar as hierarquias modeladas:

- Uma hierarquia de classes deve modelar relações “é-um-tipo-de”, ou seja, toda subclasse deve ser um tipo específico das suas superclasses.
- Uma subclasse deve suportar toda a funcionalidade (atributos, relacionamentos e operações) definida por suas superclasses, e possivelmente mais.
- A funcionalidade comum a diversas classes deve ser posicionada o mais alto possível na hierarquia.
- Classes abstratas não podem herdar de classes concretas.
- Classes que não adicionam funcionalidade devem ser eliminadas. Por adicionar funcionalidade que se dizer adicionar nova funcionalidade ou redefinir uma funcionalidade existente em uma superclasse.

Cada especialização deve ser nomeada de forma a ser auto-explicativa. Um nome apropriado para a especialização pode ser formado pelos nomes de suas generalizações, acompanhados por um nome qualificador que descreve a natureza da especialização.

As classes geradas através de generalização-especialização devem atender, além dos critérios para inclusão discutidos na seção anterior, aos seguintes critérios:

- As potenciais especializações e/ou generalizações devem estar no domínio do problema e devem fazer parte das responsabilidades do sistema; e
- Se uma classe é dita sub-classe de outra, todas as instâncias da sub-classe têm de ser também, por definição, instâncias da super-classe, isto é, tudo o que é dito sobre a super-classe (relacionamentos, atributos e operações) tem de ser válido também para a sub-classe.

Se a única distinção entre as especializações for o seu tipo, ou se as classes de especialização não adicionarem nenhuma funcionalidade, então a estrutura de generalização-especialização não é necessária.

A figura 5.3 ilustra uma hierarquia de classes dentro do contexto de uma auto-escola. Há uma pequena e importante sutileza neste exemplo que precisa ser realçada: a figura 5.3 indica explicitamente que não há instâncias da classe **Veículo** per se; as únicas instâncias (ou objetos) ocorrem no nível de especialização de **Carro**, **Moto**, **Ônibus**, ou **Caminhão**. Isso é indicado pelo nome da classe **Veículo** que está escrito em itálico, em contraste com os demais nomes de classes. Este é o padrão de notação para classes abstratas na UML.

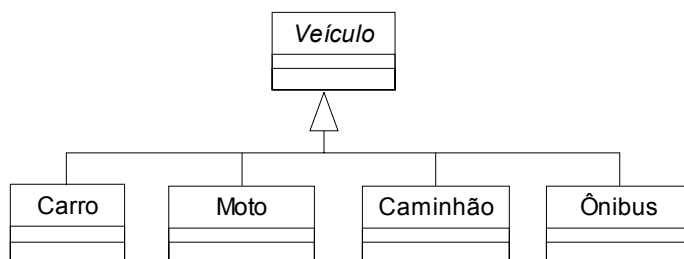


Figura 5.3 - Hierarquia de Classes no Contexto de uma Auto-Escola.

Um recurso adicional que muitas vezes facilita o entendimento das estruturas de generalização / especialização, principalmente quando há várias classificações para uma mesma classe, é indicar o critério de organização de cada hierarquia, adicionando um rótulo junto ao símbolo de classificação (triângulo).

5.3 - Identificação de Subsistemas

Um modelo de análise para uma aplicação complexa pode ter centenas de classes e dezenas de estruturas e, portanto, pode ser necessário definir uma representação concisa capaz de orientar um leitor em um modelo desta natureza. O agrupamento de classes em subsistemas serve basicamente a este propósito, podendo ser útil também para a organização de grupos de trabalho em projetos extensos. A base principal para a identificação de subsistemas é a complexidade do domínio do problema. Através da identificação e agrupamento de classes em subsistemas, é possível controlar a visibilidade do leitor e, assim, tornar o modelo mais compreensível.

Quando uma coleção de classes colaboram entre si para realizar um conjunto coeso de responsabilidades, elas podem ser vistas como um subsistema. Assim, um subsistema é uma abstração que provê uma referência para mais detalhes em um modelo de análise. Quando visto de fora, um subsistema pode ser tratado como uma caixa preta que contém um conjunto de responsabilidades e possui suas próprias colaborações [Pressman00].

O agrupamento de classes em subsistemas permite apresentar o modelo global em uma perspectiva mais alta. Este nível ajuda o leitor a rever o modelo. Além disso, constitui também um bom critério para organização da documentação.

Os casos de uso são um bom ponto de partida para o agrupamento de classes em subsistemas. Ao modelar grupos coesos de casos de uso como subsistemas, estamos relacionando as informações contidas nos modelos de classe e de casos de uso.

A UML propõe o uso de um tipo especial de diagrama de classes, onde apenas subsistemas e suas relações são representados, os chamados diagramas de pacotes, cuja notação é mostrada na figura 5.4.

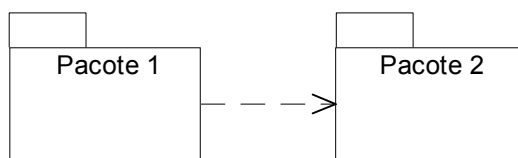


Figura 5.4 - Notação da UML para Diagramas de Pacotes.

5.4 - Identificação de Relacionamentos e Definição de Atributos

5.4.1 - Identificação de Relacionamentos

Relacionamentos são representações estáticas que modelam associações entre objetos, um dos mecanismos de estruturação de objetos. Cada classe desempenha um **papel** na associação, ao qual pode ser dado um nome. Cada papel possui também uma **cardinalidade**, que indica quantos objetos podem participar de um dado relacionamento. Em geral, a cardinalidade indica as fronteiras inferior e superior para os objetos participantes, como mostra a figura 5.5.

Um A está sempre associado a um e somente um B.	Um A está sempre associado a um ou mais Bs.	Um A está associado a zero ou um B.	Um A está associado a zero ou mais Bs.

Figura 5.5 - Notações da UML para Associações e suas Cardinalidades.

Há várias maneiras de nomear associações. Os seguidores da modelagem de dados tradicional, normalmente, têm o hábito de usar um verbo ou frase verbal de modo que o relacionamento possa ser lido como uma sentença. Na modelagem OO, contudo, há uma outra opção de nomeação, que consiste em utilizar nomes para rotular um ou mais papéis, indicando a responsabilidade dos objetos na associação. Além disso, só devemos nomear uma associação se isto melhora o entendimento que temos sobre o modelo.

Tomemos o exemplo da figura 5.6. Em uma empresa, um empregado está lotado em um departamento e, opcionalmente, pode chefiá-lo. Um departamento, por sua vez, pode ter vários empregados nele lotados, mas apenas um chefe. Sem nomear estas associações, o modelo fica confuso. Rotulando os papéis, o modelo torna-se muito mais claro. Na figura 5.6, um departamento exerce o papel de *departamento de lotação* do empregado e, neste caso, um empregado tem um e somente um departamento de lotação. No outro relacionamento, um empregado exerce o papel de *chefe* e, portanto, um departamento possui um e somente um chefe. A figura 5.7 ilustra de forma genérica a maneira de se nomear papéis em uma associação.

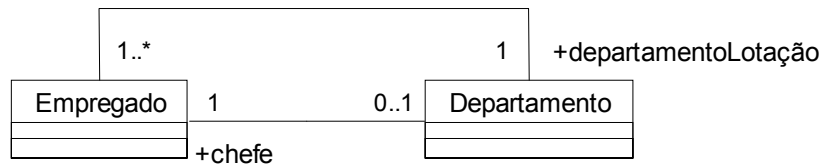


Figura 5.6 - Nomeando Associações.

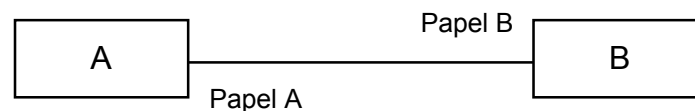


Figura 5.7 – Objetos da Classe A exercem o Papel A na associação e vice-versa.

Alguns tipos de relacionamentos merecem uma discussão adicional: os relacionamentos muitos-para muitos, os relacionamentos recursivos e os relacionamentos n-ários.

Relacionamentos muitos-para-muitos ou N:N

Relacionamentos N:N são perfeitamente legais em um modelo orientado a objetos, como mostra o exemplo da figura 5.8. Entretanto, é precisamente neste tipo de relacionamento onde mais provavelmente surgem ocorrências e eventos que precisam ser registrados e lembrados. Neste caso, uma classe do tipo “evento-lembrado” deve ser modelada, não porque esta classe deve mapear um relacionamento N:N, mas porque registrar este evento ou ocorrência é um requisito do sistema. O exemplo da figura 5.9 ilustra uma situação deste tipo.

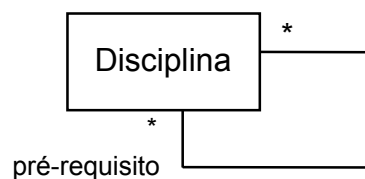


Figura 5.8 - Auto-Relacionamento N:N.

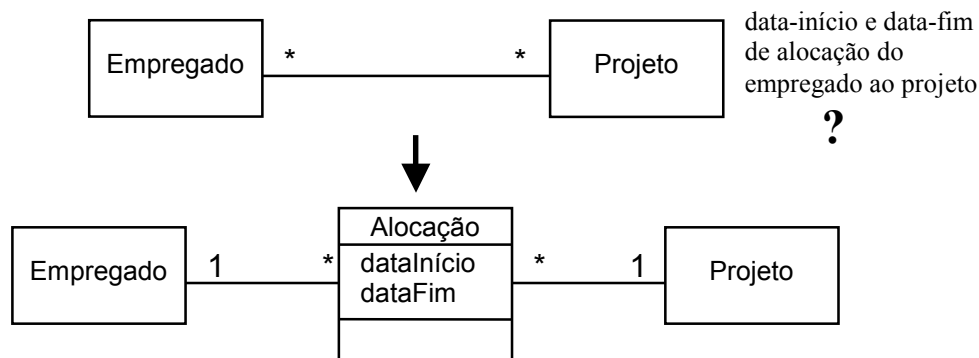


Figura 5.9: Uma Classe do Tipo “Evento-Lembrado”.

No exemplo da figura 5.9, poderíamos considerar *Alocação* como sendo uma *classe de associação*. Uma classe de associação pode ser vista como uma associação que tem propriedades de classe, isto é, atributos, operações e/ou relacionamentos. A figura 5.10 mostra este mesmo exemplo, sendo modelado como uma classe de associação, segundo a notação da UML.

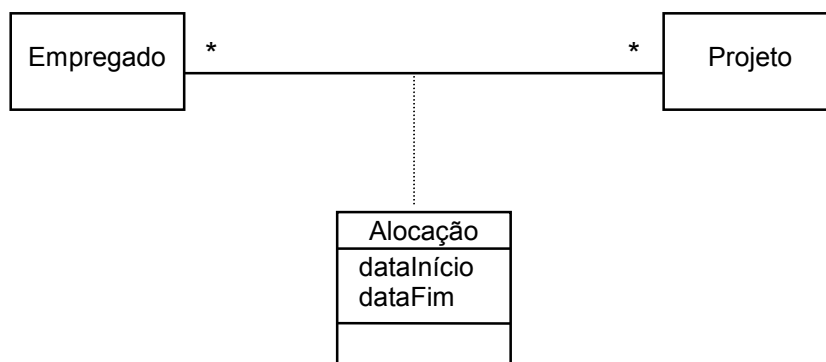


Figura 5.10 – Exemplo de Classe de Associação.

Relacionamentos Recursivos (ou Associações Unárias)

Podem ocorrer associações entre objetos de uma mesma classe. Todas as considerações feitas para relacionamentos de maneira geral são também válidas para estes casos. A figura 5.8 ilustra um relacionamento deste tipo.

Múltiplos Relacionamentos entre Objetos das Mesmas Classes

Às vezes, é necessário mais de um relacionamento entre objetos de duas classes. Nestes casos, torna-se imprescindível nomear os relacionamentos, como no exemplo da figura 5.6.

Associações *n*-árias

São associações entre três ou mais classes. Associações ternárias ou superiores são mostradas como losangos conectados às classes através de linhas, como mostra a figura 5.11. Um símbolo de classe de associação pode ser anexado ao losango por uma linha tracejada [Furlan98].

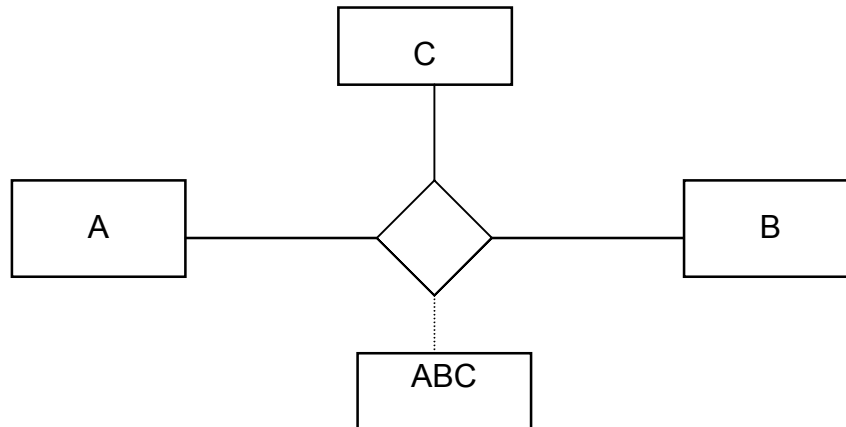


Figura 5.11 – Relacionamento Ternário.

5.4.2 - Agregação

Uma agregação é uma forma especial de associação utilizada para mostrar que um tipo de objeto é composto, pelo menos em parte, de outro em uma relação *todo/parte* [Furlan98]. Um carro, por exemplo, tem um motor e rodas como suas partes. Entretanto, muitas vezes é difícil perceber a diferença entre uma agregação e um relacionamento comum. De fato, não há uma definição única aceita por todos os métodos do que seja a diferença entre agregação e relacionamento. Ainda assim, a UML apresenta notações para agregação, como mostra a figura 5.12. É importante realçar que, uma vez que uma agregação é, na verdade, um tipo de relacionamento, cardinalidades devem ser indicadas.

Ao se utilizar a notação de agregação, espera-se que as partes “vivam” e “morram” com o todo, isto é, na criação do objeto-todo, os objetos-parte são criados e a remoção do objeto-todo deve ser aplicada em cascata às suas partes. Esta remoção em cascata é frequentemente considerada como sendo uma característica da definição de agregação, contudo, ela é imposta por qualquer papel cuja cardinalidade é 1..1. Contudo, na agregação, os objetos-parte não podem ser destruídos por outro objeto a não ser pelo objeto-todo que os criou. Esta condição não é obrigatória para objetos em uma associação comum.

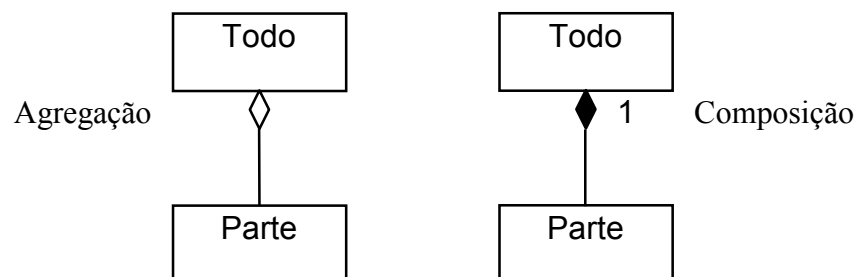


Figura 5.12 - Notações da UML para Agregação e Composição.

A UML oferece ainda notações para alguns tipos de agregação, chamadas de composição. Ao utilizar uma composição, está-se afirmando que o objeto-parte pode pertencer a apenas um todo.

5.4.3 - Definição de Atributos

Um atributo é um dado (informação de estado) para o qual cada objeto em uma classe tem o seu próprio valor. Os atributos adicionam detalhes às abstrações e são apresentados na parte central do símbolo de classe.

Atributos são muito similares a associações. De fato, conceitualmente, não há diferença entre atributos e associações. Podemos dizer que atributos representam associações entre as classes do domínio do problema e classes básicas, tais como strings, datas, inteiros e reais. Exatamente por estas classes serem básicas e ocorrerem em quase todos os domínios de problemas, elas não são representadas e, portanto, atributos são descritos como uma informação de um determinado tipo.

A estratégia para a definição de atributos envolve:

- Descoberta de atributos
- Posicionamento dos atributos em uma hierarquia de classes
- Revisão da hierarquia de classes
- Especificação dos Atributos

É bom realçar que, com o tempo, as classes do domínio de problemas permanecem relativamente estáveis, enquanto os atributos provavelmente se alteram. Atributos são bastante voláteis, em função das responsabilidades do sistema.

Descoberta de Atributos

A AOO não provê nenhuma técnica mágica para se descobrir os atributos que um objeto requer para realizar seus objetivos. A tarefa é essencialmente a mesma usada em qualquer forma de análise de sistemas: estudo do domínio da aplicação, entrevistas com o usuário, etc.

Cada atributo deve capturar um “conceito atômico”⁴ que ajude a descrever um objeto e que seja de responsabilidade do sistema. Um atributo representa uma informação de estado de um objeto que precisa ser lembrada.

Uma vez que, conceitualmente, atributos e associações são a mesma coisa, não devemos incluir na lista de atributos de uma classe, atributos representando associações. Estas já têm sua presença indicada pela linha que conecta as classes que se relacionam.

Posicionamento de Atributos

Cada atributo deve ser colocado na classe mais adequada. Para classes em uma hierarquia de generalização-especialização, atributos genéricos devem ser posicionados no topo da estrutura, de modo a serem aplicáveis a cada uma das especializações. Em outras palavras, se um atributo é aplicável a um nível inteiro de especializações, então ele deve ser posicionado na generalização correspondente. Por outro lado, se algumas vezes um atributo tiver um valor significativo, mas em outras ele não for aplicável, deve-se rever o posicionamento do atributo ou a estratégia da estrutura de generalização-especialização adotada. Esta estratégia é válida também para operações.

⁴ um único valor ou um agrupamento de valores fortemente relacionados

Revisão da Hierarquia de Classes

Inevitavelmente, o processo detalhado de designar atributos a classes conduz a um entendimento mais completo do que era possível em estágios anteriores do desenvolvimento. Assim, devemos esperar que algumas revisões de definições de classes e suas hierarquias resultem deste passo.

Especificação de Atributos

Um aspecto bastante importante na especificação de atributos é a escolha de nomes. Deve-se procurar utilizar o vocabulário padronizado para o domínio do problema, usando nomes legíveis e abrangentes.

A especificação de atributos deve incluir unidade de medida, intervalo, limite, enumeração, precisão, valor default e obrigatoriedade, entre outros. Caso valha a pena, em termos de custo-benefício, pode ser importante adicionar uma descrição breve para cada atributo.

A notação da UML para atributos é a seguinte: visibilidade nome: tipo = valorDefault, onde visibilidade é um dos seguintes símbolos:

- + , público, isto é, o atributo pode ser acessado por qualquer classe cliente;
- # , protegido, isto é, o atributo só é passível de acesso pela própria classe ou por uma de suas especializações; e
- , privado, isto é, o atributo só pode ser acessado pela própria classe.

O nome do atributo é uma sequência de caracteres de identificação, começando tipicamente, com letra minúscula. Concatena-se as demais palavras que compõem o nome, preservando-se a primeira letra de cada palavra em maiúscula [Furlan98], por exemplo, limiteDeCrédito. Pode-se também desprezar preposições. Assim, o exemplo anterior assumiria a seguinte forma: limiteCrédito.

5.5 - Determinação do Comportamento

Os Diagramas de Classes, gerados pelas atividades anteriormente descritas, representam apenas os elementos estáticos de um modelo de análise OO. É preciso analisar, ainda, o comportamento dinâmico da aplicação. Para tal, devemos representar o comportamento do sistema como uma função do tempo e de eventos específicos. Um modelo de comportamento indica como o sistema irá responder a eventos ou estímulos externos e auxilia o processo de descoberta das operações das classes do sistema.

Para apoiar a modelagem do comportamento do sistema, a UML oferece duas ferramentas:

- *Diagramas de Estados*: descrevem todos os estados possíveis pelos quais um particular objeto pode passar e suas transições, como resultados de estímulos (eventos) que atingem o objeto;
- *Diagramas de Interação*: são modelos que descrevem como grupos de objetos colaboram em um certo comportamento.

5.5.1 - Diagramas de Transição de Estados

Diagramas de Estados são uma técnica já bastante familiar para descrever o comportamento de um sistema. No entanto, no contexto do desenvolvimento OO, diagramas de transição de estados têm um caráter bastante peculiar: cada diagrama é construído para uma única classe, com o objetivo de mostrar o comportamento ao longo do tempo de vida de um objeto. Diagramas de Estado descrevem todos os possíveis estados pelos quais um objeto pode passar e as alterações dos estados como resultado de eventos (estímulos) que atingem o objeto [Furlan98]. A figura 5.13 mostra a notação básica da UML para diagramas de estado.

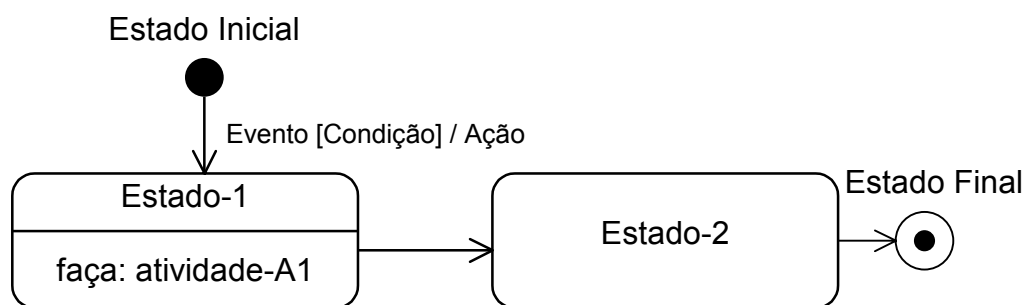


Figura 5.13 - Notação Básica da UML para Diagramas de Estados.

Estados são representados por retângulos com os cantos arredondados e transições por setas, sendo que ambos podem ser rotulados. O rótulo de um estado pode conter, além de seu nome, uma indicação de que o estado possui uma atividade associada a ele, cuja sintaxe é:

faça: atividade

O rótulo de uma transição pode ter até três partes, todas elas opcionais:

Evento [condição] / ação

Basicamente a semântica de um diagrama de estados é a seguinte: quando um **evento** ocorre, se a **condição** é verdadeira, a transição ocorre e a **ação** é realizada. O objeto passa, então, para um novo estado. Se neste estado, uma **atividade** deve ser realizada, ela é iniciada.

O fato de uma transição não possuir um evento associado, indica que a transição ocorrerá tão logo a atividade associada ao dado estado tiver sido concluída, desde que a condição seja verdadeira.

Quando uma transição não possuir uma condição associada, então ela ocorrerá sempre quando o evento ocorrer.

Embora ambos os termos ação e atividade denotem processos, tipicamente implementados por métodos da classe, eles não devem ser confundidos. Ações são associadas a transições e são consideradas processos instantâneos, isto é, ocorrem muito rapidamente, não sendo passíveis de interrupção. Atividades são associadas a estados, podendo durar bastante tempo. Assim, uma atividade pode ser interrompida por algum evento.

5.5.2 - Diagramas de Interação

Uma das coisas mais difíceis de compreender e capturar em um sistema orientado a objetos é o fluxo global de controle. Diagramas de interação têm por objetivo ajudar a visualizar esta sequência.

Tipicamente, um diagrama de interação captura o comportamento de um grupo de objetos em um caso de uso isolado, mostrando um número de objetos-exemplos e as mensagens que são passadas entre esses objetos no contexto do caso de uso [Furlan98]. Há dois tipos de diagramas de interação: diagramas de seqüência e diagramas de colaborações.

Diagramas de Seqüência

Em um diagrama de seqüência, um objeto é mostrado como um retângulo com uma linha vertical pontilhada anexada. Esta linha é chamada de *linha de vida* do objeto e representa a “vida” do objeto durante a interação. Cada mensagem é representada por uma seta cheia entre as linhas de vida de dois objetos. A ordem na qual as mensagens ocorrem é mostrada do alto para o pé da página. Cada mensagem é rotulada com pelo menos o nome da mensagem. Adicionalmente, pode incluir também seus argumentos e alguma informação de controle.

Informações de controle podem ser uma *condição* ([condição = “verdadeira”]), indicando que a mensagem só é enviada se a condição for verdadeira, e um *marcador de iteração* (*), denotando que a mensagem é enviada várias vezes para múltiplos objetos receptores, como no caso de uma iteração sobre uma coleção de objetos.

O diagrama inclui ainda, um *retorno*, indicando o retorno de uma mensagem e não uma nova mensagem. Para diferenciar retornos são simbolizados por uma seta não sólida. A figura 5.14 sumariza a notação básica da UML para diagramas de seqüência.

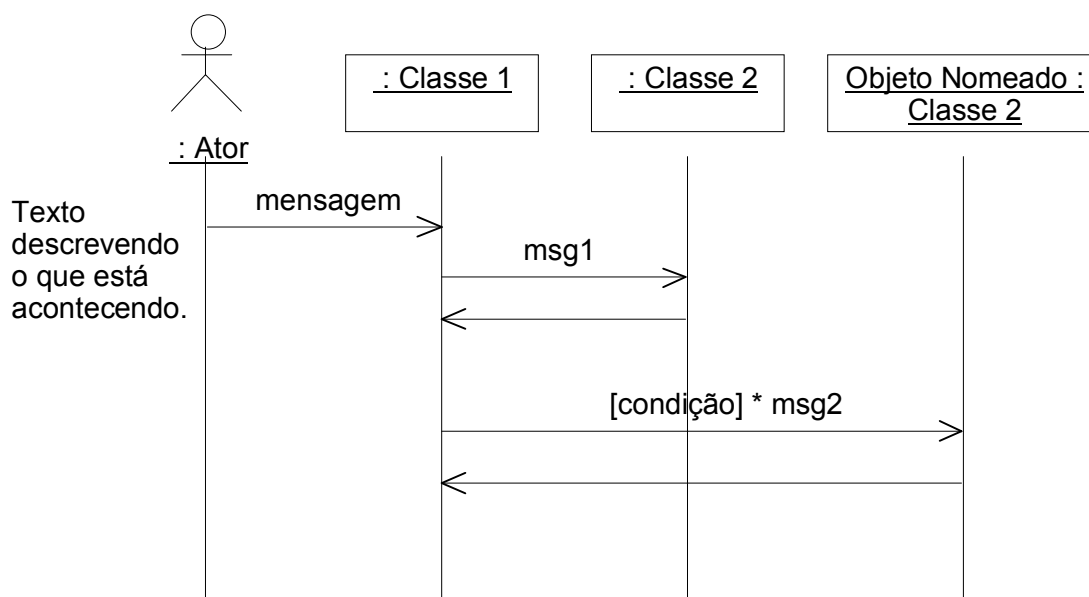


Figura 5.14 - Notação da UML para Diagramas de Seqüência.

Uma técnica bastante útil para facilitar o entendimento de um diagrama de seqüência consiste em inserir descrições textuais do que está acontecendo ao longo do lado esquerdo do diagrama de seqüência. Isto envolve a redação de um bloco de texto alinhado com a mensagem apropriada no diagrama.

Diagramas de Colaborações

Um diagrama de colaborações tem o mesmo propósito que um diagrama de sequência e, portanto, ambos são similares, ou seja, trabalham com as mesmas abstrações (objetos-exemplos, mensagens e seqüências).

Em um diagrama de colaborações, os objetos-exemplos são mostrados como ícones e mensagens como setas. A seqüência, por sua vez, é indicada por uma numeração das mensagens. A figura 5.15 mostra a notação básica da UML para diagramas de colaborações.

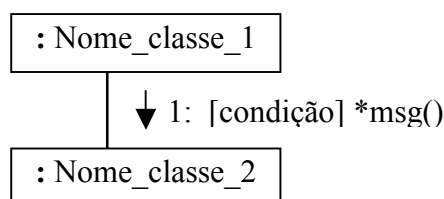


Figura 5.15 - Notação Básica da UML para Diagramas de Colaborações.

A numeração de mensagens torna a visualização da seqüência mais difícil do que nos diagramas de seqüência. Entretanto, este layout permite mostrar mais facilmente outras coisas. A UML adota um esquema de numeração decimal (1, 1.1, 1.1.1, 2, ...) com o objetivo de tornar mais claro que operação está chamando que outras operações, embora possa ser difícil ver a seqüência global.

Tanto diagramas de seqüência quanto diagramas de colaborações não são muito apropriados para representar processos com muito comportamento condicional e repetitivo. Para estes casos, ainda que seja possível utilizar condições e marcadores de iterações, é melhor utilizar diagramas separados para cada cenário.

Deve-se ressaltar, ainda, que os diagramas de interação são bons para mostrar colaborações entre objetos em um caso de uso. Para observar o comportamento de um objeto isolado ao longo de vários casos de uso, deve-se utilizar um diagrama de transição de estados.

5.6 - Definição das Operações

Uma vez estudado o comportamento do sistema, tem-se uma base para a definição das operações das classes que compõem o sistema. Na notação da UML, operações são apresentadas na seção inferior do símbolo de classe, com a seguinte sintaxe:

visibilidade nome(lista_de_parâmetros): tipo_de_retorno

onde a visibilidade tem a mesma sintaxe e semântica definida para atributos.

Normalmente, operações que simplesmente manipulam atributos não são representadas, já que podemos deduzir que elas existem, tais como:

- *Operações que obtêm valores de atributos:* apenas retornam um valor de um atributo e não fazem mais nada;
- *Operações que colocam valores em atributos:* apenas colocam um valor em um atributo e não fazem mais nada.

Além dessas, operações para criar uma nova instância da classe, selecionar e eliminar uma instância da classe também não são normalmente mostradas no diagrama de classes. Entretanto, afora os sistemas muito simples, outras operações devem ser capturadas, entre elas:

- operações associados às mensagens nos diagramas de interação;
- operações associadas aos relacionamentos entre objetos nos diagramas de classes, isto é, operações para *associar* e *desassociar* instâncias específicas que se relacionam. Em outras palavras, são as operações que permitem que um relacionamento seja estabelecido ou desfeito;
- operações sugeridas pelos diagramas de estados.

Especificação das Operações

Finalmente, as operações podem ser descritas, detalhando-se o comportamento das classes. Cada operação pode dar origem a um ou mais métodos a serem executados quando uma mensagem for recebida. Se os objetos tiverem sido bem definidos, então cada método será pequeno e altamente coeso. Assim, é possível descrever uma operação compactamente em alguma notação semi-formal, como um “português-estruturado”.

Um Recurso Adicional - Contratos

Um recurso muito útil, empregado pelo método CRC [Wirfs-Brock90] é a definição de *contratos*. Um contrato define um conjunto de requisições que um cliente pode fazer a um servidor, com a garantia de que o servidor é capaz de respondê-las. Agrupar funcionalidades em contratos auxilia o entendimento de um modelo e estabelece claramente as responsabilidades de uma classe.

Uma classe pode suportar vários contratos, apesar de ser bastante comum encontrar classes que suportam apenas um único contrato. Cada responsabilidade⁵ de uma classe pode ser parte de um contrato, mas nem todas responsabilidades o são. Algumas responsabilidades representam um comportamento que a classe deve apresentar mas que não pode ser requisitado por objetos de outras classes. Tais responsabilidades são ditas *responsabilidades privadas*.

As seguintes diretrizes ajudam a determinar que responsabilidades pertencem a que contratos:

- *Agrupe funcionalidades usadas pelos mesmos clientes*. Um contrato representa um conjunto coeso de responsabilidades. Uma maneira de encontrar responsabilidades coesas é procurar por funcionalidades que serão utilizadas sempre pelos mesmos clientes.
- *Maximize a coesão das classes*. Assim como um contrato deve ser composto de um conjunto coeso de responsabilidades, uma classe deve suportar um conjunto coeso de

⁵ Responsabilidades incluem tanto a informação mantida pelas instâncias da classe (estado dos objetos) quanto as ações que essas instâncias podem executar (comportamento dos objetos) [Wirfs-Brock90].

contratos. Maximizar a coesão tende a minimizar o número de contratos suportados por uma classe.

- *Minimize o número de contratos.* Sem violar o princípio de coesão dos contratos, a melhor maneira de reduzir o número de contratos é procurar por funcionalidades similares que possam ser generalizadas, permitindo, assim, que essas, e os contratos dos quais elas são partes, possam ser movidos para pontos mais altos na hierarquia de classes. Deste modo, um contrato pode ser definido somente em uma classe, a superclasse, e as várias classes que o suportam podem herdá-lo da superclasse.

Uma estratégia simples para a definição de contratos, começa pela definição dos contratos das classes localizadas no topo de suas hierarquias. Novos contratos só precisam ser definidos para as subclasses que adicionam nova funcionalidade, a ser usada por outros clientes. Assim, deve-se examinar as responsabilidades adicionadas por cada uma das subclasses, avaliando se elas representam nova funcionalidade ou se apenas definem maneiras mais específicas de expressar responsabilidades herdadas, caso em que já são parte do contrato herdado.

Referências Bibliográficas

- [Booch94] G. Booch; *Object-Oriented Analysis and Design with Applications*, 2nd edition, Benjamin/Cummings Publishing Company, Inc, 1994.
- [Coad92] P. Coad, E. Yourdon; *Análise Baseada em Objetos*, Editora Campus, 1992.
- [Coad93] P. Coad, E. Yourdon; *Projeto Baseado em Objetos*, Editora Campus, 1993.
- [Furlan98] J.D. Furlan; *Modelagem de Objetos Através da UML*; Makron Books, 1998.
- [Jacobson92] I. Jacobson; *Object-Oriented Software Engineering*, Addison-Wesley, 1992.
- [Pressman00] R.S. Pressman; *Software Engineering: A Practitioner's Approach*, 5th Edition, Mc Graw Hill, 2000.
- [Rumbaugh94] J. Rumbaugh, et alli; *Modelagem e Projetos Baseados em Objetos*, Editora Campus, 1994.
- [Wirfs-Brock90] R. Wirfs-Brock, B. Wilkerson, L. Wiener; *Designing Object-Oriented Software*, Prentice Hall, 1990.
- [Yourdon94] E. Yourdon; *Object-Oriented Systems Design: an Integrated Approach*, Yourdon Press Computing Series, Prentice Hall, 1994.

PARTE III – ANÁLISE ESSENCIAL DE SISTEMAS

6. Introdução à Análise Essencial

A etapa de análise e especificação de requisitos, geralmente chamada de análise de sistemas, é um processo de comunicação entre engenheiros de software (analistas de sistemas) e clientes/usuários do sistema, com o objetivo de definir detalhadamente o propósito e os requisitos de um software. Os requisitos de um sistema compreendem o conjunto de características que o sistema deve possuir para atingir seu propósito.

A análise de sistemas é um processo de transmissão de conhecimento e, assim sendo, envolve três etapas:

- ❑ aprendizado: aprender sobre o domínio do problema onde o sistema será inserido;
- ❑ estruturação e representação dos requisitos do sistema: consiste na modelagem do sistema propriamente dita;
- ❑ validação dos requisitos com o usuário.

Ao longo do processo, o analista enfrenta o desafio de “lidar com a complexidade”, isto é, situações complexas do mundo real devem ser entendidas e representadas de forma simples, para facilitar a compreensão e validação. Para tal, é preciso delimitar a área de estudo, subdividir o todo complexo em partes inteligíveis e gerenciáveis, extrair as características essenciais da realidade e modelar essas características para mostrar o relacionamento entre seus componentes.

A análise de sistemas é, em última instância, uma atividade de construção de modelos. Um modelo é uma representação de alguma coisa do mundo real, uma abstração da realidade, ou seja, representa uma seleção de características do mundo real, que são relevantes para o propósito com o qual o modelo foi construído.

Modelos são ferramentas fundamentais no desenvolvimento de sistemas. Sistemas são modelados para:

- ❑ possibilitar o estudo do comportamento do sistema;
- ❑ facilitar a comunicação entre os componentes da equipe de desenvolvimento (e clientes e usuários);
- ❑ possibilitar a discussão de correções e modificações com o usuário;
- ❑ formar uma documentação do sistema.

Um modelo enfatiza um conjunto de características da realidade, que corresponde à *dimensão do modelo*. Além da dimensão que um modelo enfatiza, modelos possuem níveis de abstração. O *nível de abstração* de um modelo diz respeito ao grau de detalhamento com que as características do sistema são representadas. Em cada nível há uma ênfase seletiva nos detalhes representados. No caso dos sistemas de informação, geralmente, são considerados três níveis:

- ❑ *conceitual*: considera características do sistema independentes do ambiente computacional (hardware e software) no qual o sistema será implementado. Essas características são dependentes unicamente das necessidades do usuário.
- ❑ *lógico*: características dependentes de um determinado *tipo* de sistema computacional. Essas características são, contudo, independentes de produtos específicos.

- ❑ *físico*: características dependentes de um sistema computacional específico, isto é, uma linguagem e um compilador específico, um sistema gerenciador de bancos de dados específico, o hardware de um determinado fabricante, etc.

Nas primeiras etapas do processo de desenvolvimento (levantamento de requisitos e análise), o engenheiro de software representa o sistema através de modelos conceituais. Nas etapas posteriores, as características lógicas e físicas são representadas em novos modelos.

O método de Análise Essencial de Sistemas [Pompilho95] preconiza que, de uma forma geral, um sistema deve ser modelado através de três dimensões:

- ❑ *dados*: diz respeito aos aspectos estáticos e estruturais do sistema;
- ❑ *controle*: leva em conta aspectos temporais e comportamentais do sistema;
- ❑ *funções*: considera a transformação de valores.

Em relação ao grau de abstração, a Análise Essencial considera dois níveis: o nível essencial e o nível de implementação, representados, respectivamente, pelos seguintes modelos:

- ❑ *Modelo Essencial*: representa o sistema num grau de abstração completamente independente de restrições tecnológicas.
- ❑ *Modelo de Implementação*: passa a considerar as restrições tecnológicas impostas pela plataforma de hardware e software a ser utilizada para implementar o sistema.

Podemos perceber que o modelo de implementação não corresponde a um modelo de análise propriamente dito, uma vez que considera aspectos de implementação, característica marcante da fase de projeto. De fato, na abordagem da Análise Essencial, este modelo corresponde a uma espécie de zona nebulosa entre as fases de análise e de projeto. Por considerarmos que um modelo considerando aspectos da plataforma de implementação é melhor caracterizado na fase de projeto, neste texto, não trataremos do modelo de implementação.

6.1 - Conceitos

Os conceitos introduzidos pelo método de Análise Essencial endereçavam inicialmente as duas principais dificuldades que os analistas enfrentavam com a aplicação da Análise Estruturada: a distinção entre requisitos lógicos e físicos, e a ausência de uma abordagem para particionar o sistema em partes tão independentes quanto possível, de modo a facilitar o processo de análise.

Durante muito tempo, houve grandes debates entre os profissionais de desenvolvimento de sistemas sobre por qual perspectiva se deveria começar a especificação de um sistema: pelos dados ou pelas funções? Os argumentos, igualmente válidos, exploravam considerações como:

- ✓ dados são mais estáveis que funções...,
- ✓ sem um entendimento das funções a serem desempenhadas pelo sistema, como definir o escopo e os dados necessários?

A Análise Essencial procurou estabelecer um novo ponto de partida para a especificação de um sistema: a identificação dos *eventos* que o afetam [Pompilho95].

Um dos problemas mais relevantes na especificação é como efetuar seu particionamento. A Análise Estruturada propõe um particionamento através de uma abordagem *top-down*. Embora esta seja uma boa maneira de se atacar um problema complexo – começando da visão geral e ir descendo, passo a passo, numa visão hierárquica, a níveis de detalhes cada vez maiores – na prática, esta abordagem não se mostrou eficiente como estratégia de projeto para a decomposição de sistemas. A Análise Essencial propõe uma outra forma de particionamento, a qual é baseada nos eventos, e que tem demonstrado ser mais efetiva do que a abordagem *top-down*, pois torna mais fácil a identificação das funções e entidades que compõem o sistema [Pompilho95].

A Análise Essencial de Sistemas, através da técnica de particionamento por eventos, oferece uma boa estratégia para modelar o comportamento do sistema, visando satisfazer os requisitos do usuário, pressupondo-se que dispomos de tecnologia perfeita e que ela pode ser obtida a custo zero [Xavier95].

Apesar de introduzir novos conceitos e novas abordagens, a Análise Essencial preservou todos os modelos da Análise Estruturada. De fato, embora diferentes, a melhor maneira de encarar a Análise Essencial é considerá-la uma evolução da Análise Estruturada. A seguir, os principais conceitos da Análise Essencial [McMenamim84] são apresentados.

Tecnologia Perfeita

Durante a fase de análise, o analista deve abstrair-se da tecnologia que deverá ser utilizada na implementação do sistema. Para orientar essa abstração, a Análise Estruturada recomenda que o analista, durante a fase de análise, concentre-se apenas nos aspectos lógicos do sistema, evitando pensar nos aspectos físicos. O problema dessa abordagem é que a diferença entre “aspectos lógicos e físicos” não é clara.

Partindo do princípio que os aspectos físicos de um sistema estão ligados à tecnologia de implementação, a Análise Essencial emprega uma abstração de uma tecnologia de implementação, denominada *tecnologia perfeita*, para facilitar a tarefa de identificar os detalhes lógicos do sistema. A tecnologia perfeita não possui limitações, isto é, existe um processador perfeito, capaz de executar qualquer processamento, tudo instantaneamente, sem qualquer custo, sem consumir energia, sem gerar calor, sem jamais cometer erros ou parar de funcionar, e um repositório perfeito, capaz de armazenar quantidades infinitas de dados e de ser acessado instantaneamente por qualquer processador, da forma que for mais conveniente.

Naturalmente, não existe uma tecnologia de implementação com tais características. Então, qual é a utilidade dessa abstração?

Quando o analista pensa em aspectos físicos, ele está, na verdade, tentando identificar (e resolver) as limitações de uma determinada tecnologia. Pensamentos típicos do gênero são: quanto de espaço em disco vou precisar? qual o melhor método de acesso aos dados, considerando as funções do sistema? que capacidade de processamento devo necessitar? Contudo, nenhuma dessas preocupações são próprias da fase de análise.

Considerando agora que a tecnologia que será utilizada na implementação do sistema é perfeita, todas as perguntas anteriores deixam de ter importância, isto é, não preocupam mais o analista. Assim sendo, para distinguir um requisito lógico de um requisito físico, utilizando a abstração de tecnologia perfeita, formule a seguinte pergunta ao identificar um requisito qualquer: “Para atender ao seu propósito, o sistema necessitará possuir essa capacidade ou essa característica, mesmo considerando que ele será implementado em uma tecnologia perfeita?” Se a resposta for sim, esse requisito é verdadeiro e deve ser modelado.

Requisito Verdadeiro e Requisito Falso

Uma característica ou capacidade que um sistema deve possuir para atender ao seu propósito, mesmo considerando que será implementado em uma tecnologia perfeita, é dita um *requisito verdadeiro*. O conjunto de requisitos verdadeiros compreende a *essência do sistema*.

Um *requisito falso*, por outro lado, é uma capacidade ou característica que o sistema não precisa possuir para atender ao seu propósito, caso ele disponha de uma tecnologia de implementação perfeita.

Evento e Resposta

Evento e resposta são nomes genéricos de interações entre o ambiente externo e o sistema. Um *evento* pode ser definido informalmente como um acontecimento do mundo exterior que requer do sistema uma resposta. Corresponde a alguma mudança no ambiente externo que funcionará como um *estímulo* para o sistema, isto é, o sistema deve responder a este estímulo para atender ao seu propósito. Uma *resposta* é o resultado da execução de um conjunto de ações no sistema, como consequência do reconhecimento pelo sistema de que um evento ocorreu. Uma resposta tipicamente pode ser [Pompilho95]:

- um fluxo de dados saindo do sistema para uma entidade externa;
- uma mudança de estado em um depósito de dados (inclusão, exclusão ou alteração);
- um fluxo de controle saindo de uma função para ativar uma outra.

Quando um evento ocorre, é produzido um estímulo para o sistema. Ao receber o estímulo, o sistema compreende que o evento ocorreu e ativa os processos necessários para produzir a resposta.

Os eventos são classificados em quatro tipos diferentes, dependendo da maneira como ocorrem e da natureza do estímulo que produzem [Pompilho95]:

- **Evento orientado por fluxo de dados:** é provocado por uma entidade externa, a qual envia dados para o sistema. O estímulo produzido por esse tipo de evento é a chegada de um fluxo de dados que vai ativar uma função. Os eventos externos são nomeados da seguinte forma:

(*Entidade externa que provocou o evento*) +
(*ação* – verbo na voz ativa) +
(*estímulo* – fluxo de dados enviado ao sistema)

Ex.: Cliente envia pedido.

Cliente cancela pedido.

- **Evento orientado por controle:** o estímulo provocado por este evento é a chegada ao sistema de um fluxo de controle, o qual apenas notifica o sistema da ocorrência do evento. Pode haver fluxos de dados complementares associados ao evento, mas não é através da chegada do fluxo de dados que o sistema toma conhecimento da ocorrência do evento. Esse tipo de evento pode ser nomeado de duas maneiras, tendo por base a origem do evento:

- Caso 1 – Uma entidade externa envia um comando (fluxo de controle) para o interior do sistema, ativando uma função.

(Entidade externa que provocou o evento) +
(ação – verbo na voz ativa) +
(complemento)

Ex.: Gerente solicita relação de clientes.

Diretoria autoriza o pagamento de uma fatura.

- Caso 2 – Uma função é ativada por um fluxo de controle oriundo de outra função interna.

(Sujeito) + (ação – verbo na voz passiva)

Ex.: Nível de reabastecimento do estoque de um produto é atingido.

- **Evento temporal:** é aquele em que o estímulo é a chegada ao sistema da informação de haver passado um determinado intervalo de tempo. Esses eventos estimulam as ações que o sistema tem de executar em datas previamente conhecidas, isto é, diariamente, mensalmente, etc (o tempo passa e chega o momento do sistema fazer alguma coisa). Pode haver fluxos de dados complementares associados ao evento, mas não é através da chegada destes que o sistema toma conhecimento da ocorrência do evento. Os eventos temporais podem ser nomeados como abaixo:

(É hora de) + (ação) + (complemento)

Ex.: Mensalmente, emitir relatório de vendas. ou

É hora de emitir relatório mensal de vendas.

Atividades Essenciais

São todas as tarefas que o sistema deve executar para atender completamente ao seu propósito, mesmo considerando que ele será implementado em uma tecnologia perfeita. Uma atividade essencial deve executar todo o conjunto de ações necessárias para responder completamente a um e somente um evento. As atividades essenciais subdividem-se em:

- **Atividades Fundamentais:** produzem uma informação que é parte do propósito declarado do sistema. Assim sendo, o propósito do sistema é atendido pelas atividades fundamentais, as quais produzem as respostas externas do sistema.
- **Atividades Custodiais:** criam e mantêm a memória necessária à execução das atividades fundamentais, adquirindo dados do ambiente externo ao sistema e os armazenando nos depósitos de dados. As respostas que são internas ao sistema são produzidas pelas suas atividades custodiais.

Quando uma atividade executa tarefas dos dois tipos, ela é denominada *atividade composta*. As atividades compostas produzem respostas internas e externas. Os diferentes tipos de atividade essencial estão representados na figura 6.1.

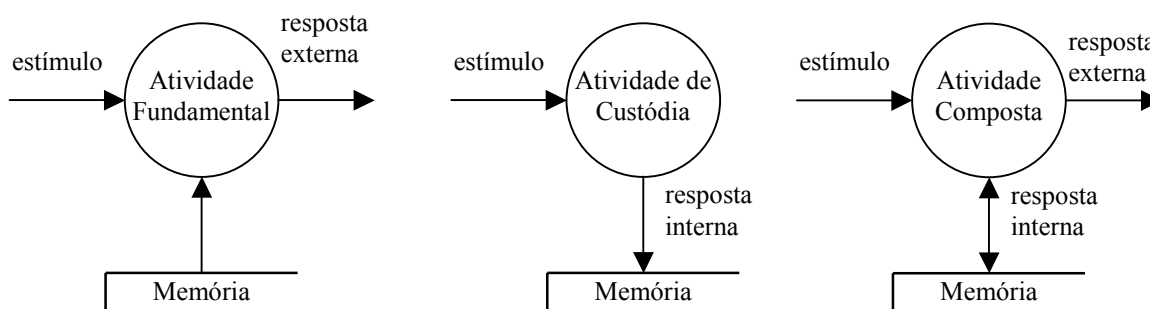


Figura 6.1 – Tipos de Atividades Essenciais.

Como as atividades essenciais respondem completamente a um e somente um evento, a comunicação entre elas será feita sempre via memória e nunca diretamente. Essa característica da comunicação entre atividades essenciais torna o *particionamento por eventos* uma abordagem adequada para dividir o problema em partes independentes.

Memória Essencial

Consiste no conjunto mínimo de dados que deve ser armazenado pelo sistema, para atender ao seu propósito, considerando que ele será implementado em uma tecnologia perfeita.

O modelo normalmente utilizado para modelar a memória essencial é o Modelo de Entidades e Relacionamentos (MER). Nos DFDs, a memória essencial é representada pelos depósitos de dados. Para derivar os depósitos de dados do DFD a partir do MER, utilize a seguinte correspondência: cada entidade e relacionamento com atributos do MER será um depósito de dados do DFD.

Para manter a abstração da tecnologia perfeita consistente, os depósitos de dados não armazenam chaves estrangeiras (atributos determinantes transpostos entre entidades) para representar um relacionamento entre entidades, pois essa é uma característica específica dos bancos de dados relacionais, uma tecnologia nada perfeita. Lembre-se que, na fase de análise, a tecnologia de implementação ainda não foi selecionada e deve ser considerada perfeita.

Para indicar que o relacionamento entre entidades existe, sem no entanto definir como ele será implementado, a representação dos acessos das atividades de custódia à memória essencial deve obedecer à seguinte regra geral: ao criar ou excluir um relacionamento ou uma entidade que participa de um relacionamento, mostre o acesso aos depósitos de dados que correspondem ao relacionamento e às entidades que participam do relacionamento. A figura 6.2 mostra a representação gráfica desses acessos.

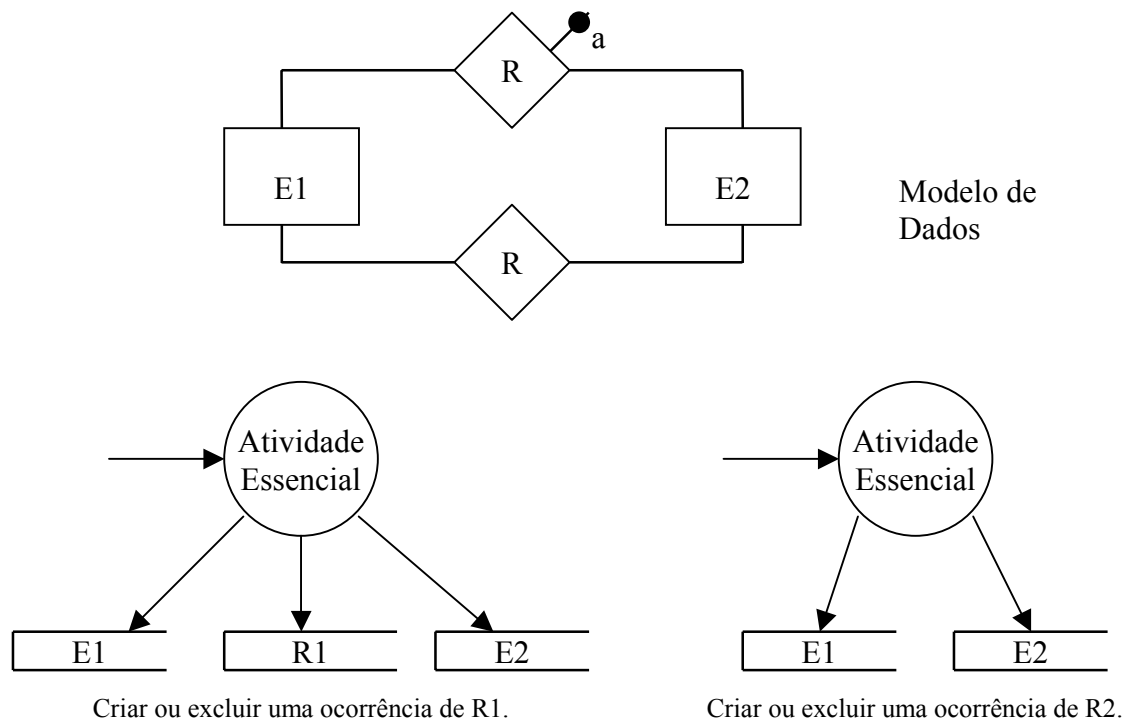


Figura 6.2 – Acessos a depósitos de dados.

6.2 - Especificação da Essência do Sistema (Referência: Capítulo 17 [Pompilho95])

A Análise Essencial sugere a construção de dois modelos principais, o modelo essencial e o modelo de implementação. Conforme discutido anteriormente, entendemos que apenas o modelo essencial deve ser objeto da fase de análise e, assim, discutiremos apenas a especificação da essência do sistema.

A especificação da essência do sistema, produto da fase de análise, é composta de dois modelos, como mostra a figura 6.3:

- ❑ Modelo Ambiental: define a fronteira entre o sistema e o resto do mundo.
- ❑ Modelo Comportamental: define o comportamento das partes internas do sistema necessário para interagir com o ambiente.

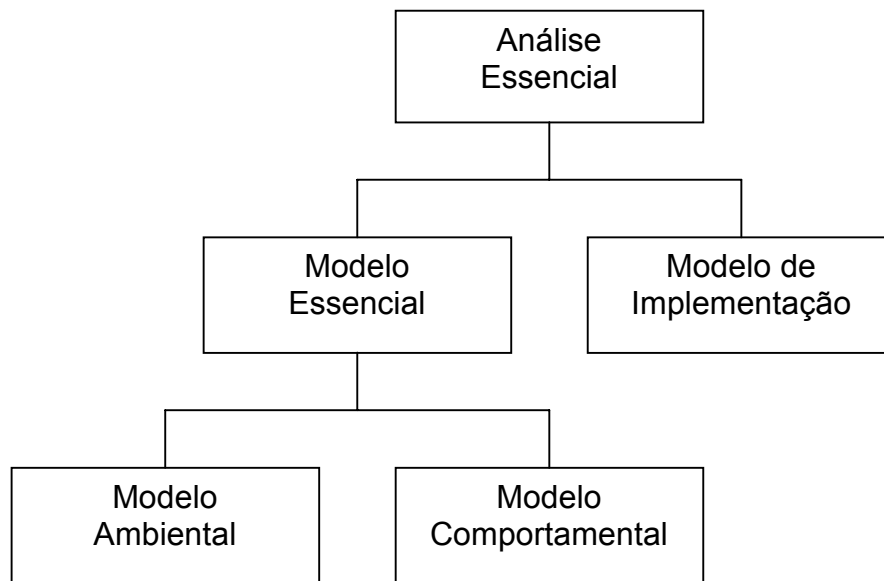


Figura 6.3 – A Análise Essencial e seus Modelos.

6.2.1 - O Modelo Ambiental (Referência: Seção 17.1 [Pompilho95])

Representa o que o sistema deve fazer para atender ao ambiente. É composto dos seguintes produtos:

- ❑ **Propósito do Sistema:** enuncia a finalidade do sistema. Pode ser acompanhado de uma breve descrição do contexto do sistema (mini-mundo).
- ❑ **Lista de Eventos:** lista de eventos aos quais o sistema deve responder. Deve conter, pelo menos, o nome do evento, o estímulo e a resposta externa do sistema.
- ❑ **Diagrama de Contexto:** representa o sistema como um único processo e suas interações com o ambiente. Pode ser acompanhado de um dicionário de dados.

A declaração de propósito (objetivos) do sistema deve ser elaborada em poucas frases, simples e precisas, em linguagem destituída de vocabulário técnico, de modo a ser entendida pelos usuários do sistema e pela administração da empresa, em geral. Não deve fornecer detalhes sobre como o sistema deverá operar.

A elaboração da lista de eventos é o passo principal desta etapa do desenvolvimento, uma vez que os eventos constituem a parte fundamental de um sistema. De fato, o primeiro passo na especificação de um sistema é identificar a quais eventos do mundo exterior ele deverá ocorrer. Esta atividade, denominada Análise de Eventos, é muito bem explorada no Capítulo 15 de [Pompilho95].

Uma vez definidos os eventos, é possível construir o Diagrama de Contexto do sistema, mostrando como ele responde a todos os eventos externos relevantes.

Finalmente, pode ser útil elaborar uma descrição de como o sistema responderá a cada um dos eventos identificados na Lista de Eventos.

6.2.2 – O Modelo Comportamental (Referência: Seção 17.2 [Pompilho95])

Representa o que o interior do sistema deve fazer para atender ao ambiente. Deve conter os seguintes produtos:

- ❑ **Diagrama de Entidades e Relacionamentos**
- ❑ **Diagramas de Fluxos de Dados Particionados por Eventos:** Para cada evento do sistema, deve ser construído um DFD. Assim, a quantidade de diagramas deve ser equivalente ao número de eventos na lista.
- ❑ **Diagramas de Transição de Estados:** Representa o comportamento das entidades e relacionamentos com atributos ao longo do tempo. Será construído um DTE para cada entidade ou relacionamento com atributo do DER que possuir comportamento significativo, isto é, possuir mais de um estado ao longo de seu ciclo de vida.
- ❑ **Diagramas de Fluxos de Dados Organizados em Níveis Hierárquicos:** representa os processos em níveis hierárquicos, a partir do diagrama zero. Os processos do diagrama zero são obtidos através do agrupamento de atividades essenciais dos DFDs particionados por eventos. Um critério de agrupamento bastante razoável é considerar o grau de coesão e acoplamento entre atividades essenciais. As seguintes heurísticas podem ser utilizadas, em conjunto ou em separado:
 - Procurar agrupar em um único processo todas as atividades essenciais que acessam um determinado depósito de dados, verificando se o processo resultante desse agrupamento é adequado para representar uma das funções do sistema.
 - Agrupar todas as atividades de custódia referentes a um mesmo depósito de dados.
 - Procurar identificar uma função do sistema, agrupando atividades essenciais que interagem com uma mesma entidade externa.
 - Representar no DFD-zero, um processo para cada uma das funções do negócio. Agrupar as atividades essenciais aos processo para os quais as suas ações mais contribuem.

Usando esta abordagem para a construção de diagramas hierárquicos, adotamos uma estratégia *middle-out* (do meio para fora), onde, a partir dos eventos, estabelecemos atividades essenciais (meio) para depois agrupá-las em níveis superiores (para cima) e, em seguida, especificá-las e, se necessário, explodi-las (para baixo).

- ❑ **Dicionário de Dados:** descreve os dados representados no MER, nos DFDs e nos DTEs.
- ❑ **Especificação da Lógica dos Processos:** descreve a lógica dos processos do DFD que não foram detalhados em diagramas de nível inferior (lógica dos processos primitivos).

Como podemos perceber, a Análise Essencial faz uso praticamente das mesmas técnicas de modelagem da Análise Estruturada, a saber a Modelagem de Dados (utilizando modelos de Entidades e Relacionamentos), a Modelagem Funcional (utilizando Diagramas de

Fluxo de Dados – DFDs) e a Modelagem de Controle (utilizando Diagramas de Transição de Estados). Isso é bastante natural, já que a Análise Essencial é, de fato, uma extensão da Análise Estruturada.

Na realidade, a principal diferença entre a Análise Essencial e a Análise Estruturada está na estratégia para atacar o problema: a primeira defende uma abordagem baseada em eventos, onde a Análise de Eventos passa a ser um passo fundamental, a segunda é baseada apenas na decomposição *top-down* da funcionalidade do sistema.

A figura 6.4 apresenta de forma sintética a organização do modelo essencial.

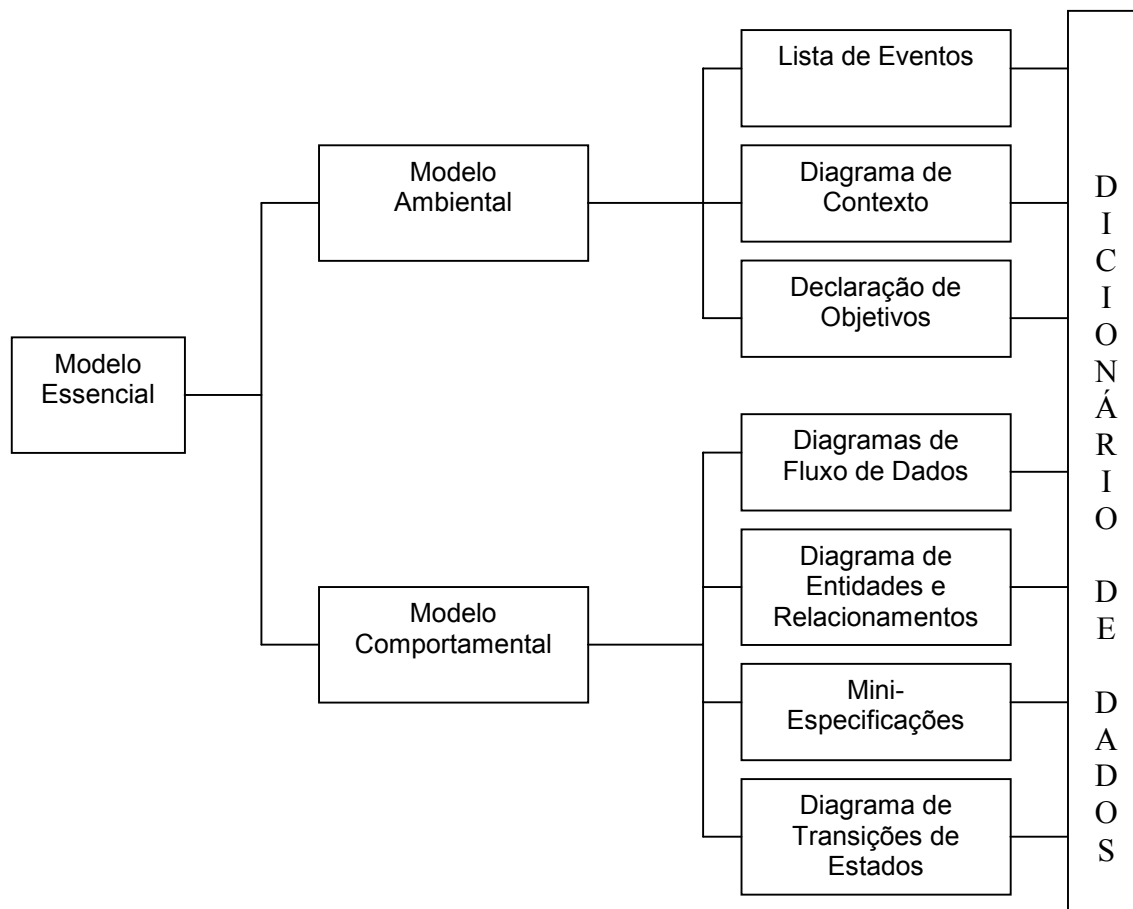


Figura 6.4 – Organização do Modelo Essencial.

Referências Bibliográficas

- [McMenamim84] S.M. McMenamim, J.F. Palmer. *Análise Essencial de Sistemas*. McGraw-Hill, São Paulo, 1984.
- [Pompilho95] S. Pompilho. *Análise Essencial: Guia Prático de Análise de Sistemas*. IBPI Press, Editora Infobook, Rio de Janeiro, 1995.
- [Xavier95] C.M.S. Xavier, C. Portilho. *Projetando com Qualidade a Tecnologia de Sistemas de Informação*. Livros Técnicos e Científicos Editora, 1995.
- [Yourdon90] E. Yourdon. *Análise Estruturada Moderna*. Editora Campus, 1990.

7. Modelagem de Dados

A primeira atividade realizada no processo de construção do Modelo Comportamental da Análise Essencial de Sistemas deve ser a modelagem de dados e, para tal, o modelo de Entidades e Relacionamentos (ER) é utilizado. O modelo ER é uma técnica top-down de modelagem conceitual, utilizada para representar os dados a serem armazenados em um sistema de informação, tendo sido desenvolvida originalmente para dar suporte ao projeto de bancos de dados [Chen90] [Setzer87].

Basicamente, o modelo ER representa as *entidades* (coisas) e os *relacionamentos* (fatos) do mundo real, que um sistema de informação precisa simular internamente.

7.1 - Conceitos Básicos

Entidades: são representações abstratas de “coisas” do mundo real que temos interesse em monitorar o comportamento. Podem ser a representação de um ser, um objeto, um organismo social, etc.

Conjuntos de Entidades: são grupos de entidades que têm características semelhantes. São representados graficamente por retângulos, como mostra a figura 7.1.

Ex: Livros, Clientes, Funcionários.



Figura 7.1 – Representação Gráfica de Conjuntos de Entidades.

Os conjuntos de entidades são substantivos e perduram no tempo. Cada elemento de um conjunto de entidades só ocorre uma única vez e a ordenação do conjunto é irrelevante.

A princípio são representados em um conjunto de entidades, todos os elementos do mundo real referidos pelo conjunto.

Ex: LIVROS = todos os livros de uma biblioteca.

FUNCIONÁRIOS = todos os funcionários de uma empresa, ...

Para descrevermos conjuntos de entidades, podemos utilizar as seguintes diretrizes:

- critérios para inclusão;
- critérios para exclusão;
- contexto (ilustre como ele é utilizado no sistema);
- exemplos.

Se não quisermos utilizar todas as diretrizes apresentadas, devemos optar pela utilização de descrições com base nos critérios para inclusão.

Para estabelecermos uma padronização, usaremos nomes de conjuntos de entidades sempre no plural e escritos em letras maiúsculas. No entanto, isto não representa efetivamente uma regra.

Atributos: descrevem características ou propriedades relevantes de um conjunto de entidades.

O conjunto de atributos deve ser:

- completo: deve abranger todas as informações pertinentes.
- fatorado: cada atributo deve capturar um aspecto em separado.
- independente: os domínios de valores de atributos devem ser independentes uns dos outros.

Atributos podem ser de dois tipos:

- Atributos Descritivos: descrevem características intrínsecas do objeto. Ex: sexo, altura, nacionalidade, etc ...
- Atributos Nominativos: nomes e rótulos arbitrários dados aos objetos. Ex: nome, matrícula, etc ...

Sobre atributos são pertinentes ainda as seguintes considerações:

- Atributo Monovalorado: atributo que assume um único valor para cada elemento do conjunto de entidades.
Ex: matrícula, nome, data-admissão, ... de FUNCIONÁRIOS: Cada funcionário possui uma única matrícula, um único nome, etc ...
- Atributo Multivalorado: atributo que pode assumir vários valores para cada um dos elementos do conjunto de entidades. São representados com um asterisco (*) associado.
Ex: telefone* de FUNCIONÁRIOS: Um mesmo funcionário pode ter mais que um telefone.
- Valor vazio para um atributo: quando para algum ponto do conjunto de entidades não existe um valor para aquele atributo, ou ele ainda não é conhecido.
Ex: telefone* de FUNCIONÁRIOS: Um funcionário qualquer pode não ter nenhum telefone, ou em um dado momento ele não ser conhecido.
- Atributo Composto: atributo composto de um ou mais sub-atributos.
Ex: endereço, composto de rua, número, complemento, bairro, cidade, estado e cep.
- Identificadores ou Atributos Determinantes: conjunto de um ou mais atributos que identificam univocamente um elemento do conjunto de entidades. Os atributos determinantes deverão ser sublinhados.

Atributos também descrevem características de relacionamentos (atributos de relacionamentos). Todas as considerações feitas até então são válidas, sendo que uma discussão sobre características típicas destes atributos foi propositalmente postergada, visando uma melhor compreensão.

A figura 7.2 ilustra a representação gráfica para atributos. Ainda que esta notação possa ser empregada, de maneira geral, atributos são representados apenas no dicionário de dados para evitar modelos complexos e de difícil leitura.

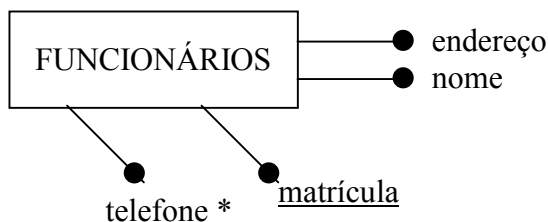


Figura 7.2 – Representação gráfica para atributos.

Relacionamento: é uma abstração de uma associação entre duas ou mais entidades.

Relacionamento Binário: é uma representação abstrata da associação de duas entidades.

Conjunto de Relacionamentos: é um subconjunto do produto cartesiano dos conjuntos de entidades envolvidos.

Ex: O mundo real nos conta que:

Funcionários são lotados em Departamentos.

Alunos cursaram Disciplinas.

Fornecedores fornecem Materiais.

É importante notar que todos os relacionamentos binários possuem uma leitura inversa:

Departamentos lotam Funcionários.

Disciplinas foram cursadas por Alunos.

Materiais são fornecidos por Fornecedores.

Algumas correntes pregam o uso de um nome que abstraia a direção da leitura.

Alunos cursaram Disciplinas. \Rightarrow Realizações

Fornecedores fornecem Materiais. \Rightarrow Fornecimentos

Neste texto, entretanto, adotaremos a seguinte notação: Um relacionamento será representado por um losango com um verbo para indicar a ação e uma seta para informar o sentido de leitura, como mostra a figura 7.3.

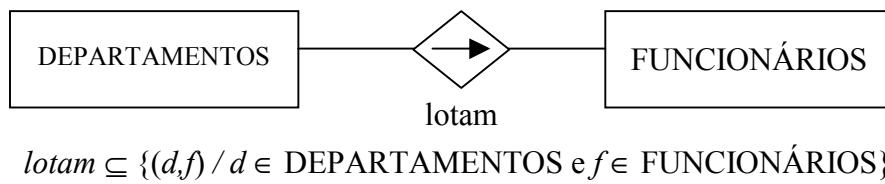


Figura 7.3 – Representação gráfica para relacionamentos.

É importante frisar que, entre duas entidades, pode existir mais de um tipo de relacionamento, como mostra o exemplo da figura 7.4.

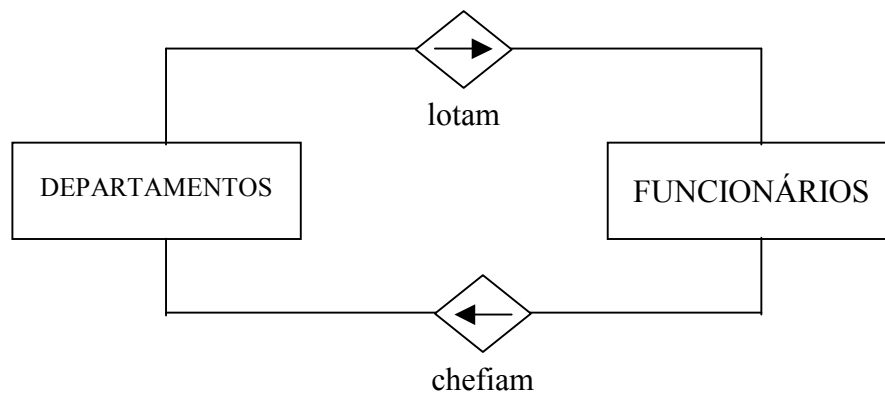


Figura 7.4 – Dois tipos de relacionamentos entre entidades.

Além disso, uma entidade pode participar de relacionamentos com quaisquer outras entidades do modelo, inclusive com ela mesma, como mostra a figura 7.5.

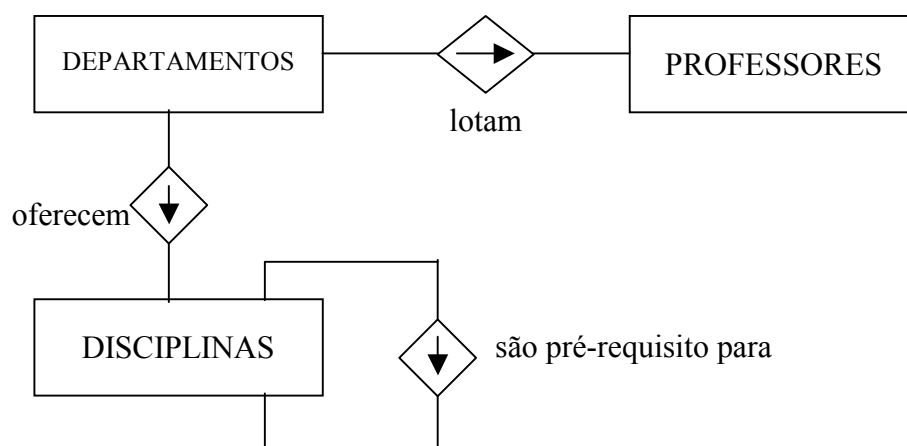


Figura 7.5 – Exemplo de modelo ER.

7.2 - Restrições de Integridade ou Leis de Consolidação

Restrições de integridade (ou leis de consolidação) são restrições do mundo real que devem ser expressas para manter a integridade do modelo. Devemos identificar leis que regem: os possíveis relacionamentos e os valores dos atributos.

7.2.1 - Restrições de Integridade em Relacionamentos

Um conjunto de relacionamentos é um subconjunto do produto cartesiano das entidades envolvidas. É necessário, portanto, descrever de forma mais apurada qual é este subconjunto. Isto é feito via Restrições de Integridade ou Leis de Consolidação, que devem ser observadas, sendo que elas podem ser de três tipos:

- **Cardinalidade:** indica os números mínimo (cardinalidade mínima) e máximo (cardinalidade máxima) de associações possíveis em um relacionamento.

Ex: Um professor tem que estar lotado em um e somente um departamento, enquanto um departamento deve ter no mínimo 13 professores e no máximo um número arbitrário (N). Esta restrição imposta pelo mundo real, deve ser considerada no modelo de dados através da cardinalidade, como mostra a figura 7.6.

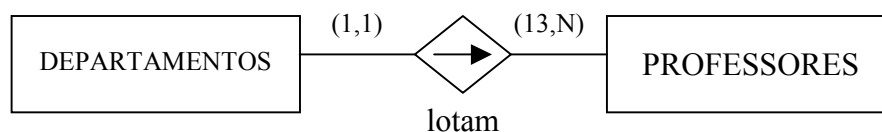


Figura 7.6 – Representação de cardinalidades em relacionamentos.

- **Repetição:** indica quantas vezes os mesmos dois elementos de conjuntos de entidades podem ser associados.

Ex: Um aluno não pode cursar a mesma disciplina mais do que 3 vezes.

- **Dependência:** um tipo de relacionamento pode ser restringido por um outro relacionamento, ou depender de suas associações anteriores.

Ex: Um aluno não pode matricular-se em uma disciplina que ainda não tenha cumprido seus pré-requisitos.

Um empregado não pode ser colocado em um cargo cujo salário seja inferior ao do seu cargo atual.

Restrições de Integridade de Dependência e Repetição não são representadas no diagrama, como ocorre com a Cardinalidade, e devem ser descritas em um dicionário do projeto.

7.2.1.1 - Tipos de Relacionamentos

Correspondem a uma classificação baseada na cardinalidade máxima dos relacionamentos. Para ilustrar, tomemos um relacionamento R entre dois conjuntos de entidades A e B, como mostra a figura 7.7.

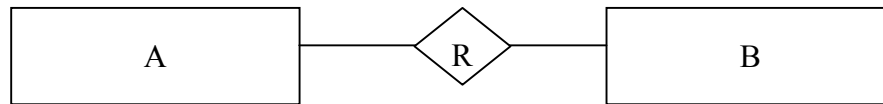


Figura 7.7 – Relacionamento R entre dois conjuntos de entidades A e B.

- **Relacionamento 1:1:** cada elemento de A ou de B pode aparecer no máximo em um único par de R, como mostra o exemplo da figura 7.8.

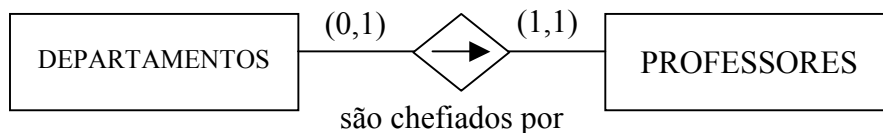


Figura 7.8 – Relacionamento 1:1.

- **Relacionamentos 1:N ou N:1:** cada elemento de B pode aparecer no máximo em um único par de R, enquanto cada elemento de A pode ocorrer em um número qualquer de pares, como ilustra o exemplo da figura 7.9.

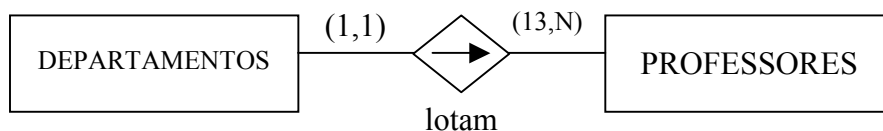


Figura 7.9 – Relacionamento 1:N.

- **Relacionamentos N:N:** cada elemento de A ou de B pode aparecer em um número não determinado de pares de R, como ilustra o exemplo da figura 7.10.

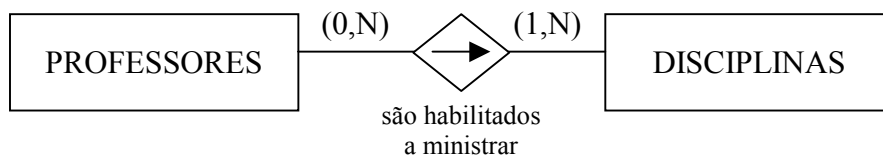


Figura 7.10 – Relacionamento N:N.

A figura 7.11 mostra um exemplo com os vários tipos de relacionamentos.

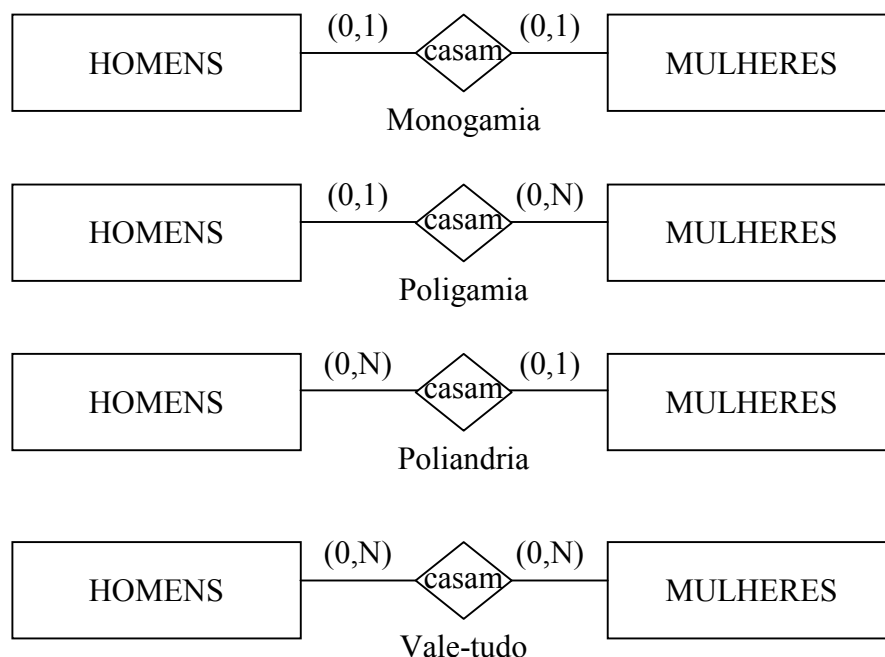


Figura 7.11 – Tipos de Relacionamentos.

7.2.1.2 - Relacionamentos Totais e Parciais

Estes tipos de relacionamentos correspondem a uma classificação relacionada com a cardinalidade mínima do relacionamento. Tomando o exemplo da figura 6.7, temos que:

- **Relacionamento Total:** dizemos que R é um relacionamento total em A se e somente se: $\forall a \in A, \exists b \in B / (a, b) \in R$, isto é, todo elemento de A tem de participar obrigatoriamente de R e, conseqüentemente, a cardinalidade mínima de R em relação a A é 1.
- **Relacionamento Parcial:** dizemos que R é um relacionamento parcial em A se e somente se: $\exists a \in A / \forall b \in B / (a, b) \notin R$, isto é, nem todo elemento de A participa de R e, conseqüentemente, a cardinalidade mínima de R em relação a A é 0.

No exemplo da figura 7.12, dizemos que o relacionamento *lotam* é total em relação a DEPARTAMENTOS e a FUNCIONÁRIOS, pois em ambos os casos a cardinalidade mínima é 1.

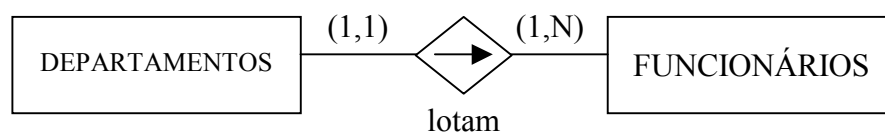


Figura 7.12 – Relacionamento Total.

No exemplo da figura 7.13, dizemos que o relacionamento *cursam* é total em relação a ALUNOS e parcial em relação a CURSOS, pois no segundo caso, a cardinalidade mínima é 0.

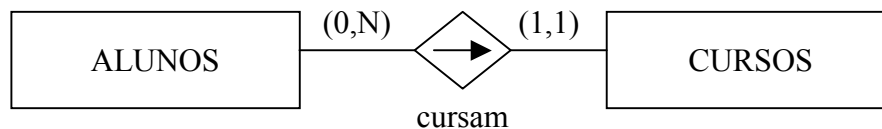


Figura 7.13 – Relacionamento parcial.

Vale a pena observar que a simbologia utilizada na representação de entidades, relacionamentos e restrições de integridade de cardinalidade não é padronizada, isto é, não foi definido um padrão único a ser seguido por todos que utilizam o Modelo ER.

7.2.2 - Restrições de Integridade sobre o Domínio dos Atributos

Ainda visando manter a integridade do modelo de dados, devemos descrever no dicionário de projeto restrições de integridade (ou leis de consolidação) que regem os valores dos atributos, isto é, o conjunto de valores que um atributo pode assumir. Esta tarefa deve ser feita utilizando-se dos seguintes recursos:

- **enumeração:** lista explícita de valores.
Ex: Estado Civil : solteiro, casado, desquitado, divorciado e viúvo.
- **normas de aceitação:** regras para se identificar se o valor é válido ou não.
Ex: Nome : qualquer conjunto de caracteres alfanuméricos, começado por uma letra.
- **intervalo:** descrição de um subconjunto de um intervalo conhecido.
Ex: Mês: de 1 até 12.

Uma vez estabelecido o domínio, é interessante determinar valores possíveis e prováveis, isto é, alguns valores, apesar de poderem ocorrer, é pouco provável que ocorram, dependendo do contexto. Por exemplo, com relação ao atributo idade de um empregado, o valor 81 é um valor possível, mas será que ele é um valor provável em um contexto de uma empresa de mineração de carvão ?

Outros aspectos que devem ser considerados na descrição dos atributos são:

- **obrigatoriedade:** estabelecer se um determinado atributo pode ter um valor nulo a ele associado.
Ex: Telefone : opcional. Nome : obrigatório.
- **dependência:** Os valores que um atributo pode assumir, muitas vezes, são dependentes dos valores de outros atributos. Neste caso é importante relacionar no dicionário de projeto como se dá esta dependência.
Ex: O valor do atributo *dia* depende fundamentalmente do valor do atributo *mês*.

Além disso, os valores que um atributo pode assumir em um novo estado, muitas vezes, dependem dos valores de estados anteriores. Neste caso, é importante relacionar como deve se dar a transição de um estado para outro.

Ex: Estado Civil: não pode passar de solteiro para viúvo.

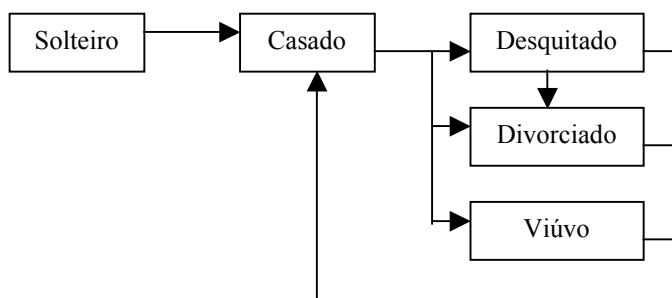


Figura 7.14 – Possíveis mudanças para o atributo *estado civil*.

7.3 - Outras Considerações sobre Atributos

7.3.1 - Atributos de Relacionamentos

Assim como as entidades, os relacionamentos também podem possuir atributos. Atributos de relacionamentos são atributos que não são de nenhuma das duas entidades, mas sim do relacionamento entre elas e, em geral, estão relacionados com “protocolos” e datas, ou são resultantes de “avaliações”, tal como no exemplo da figura 7.15.

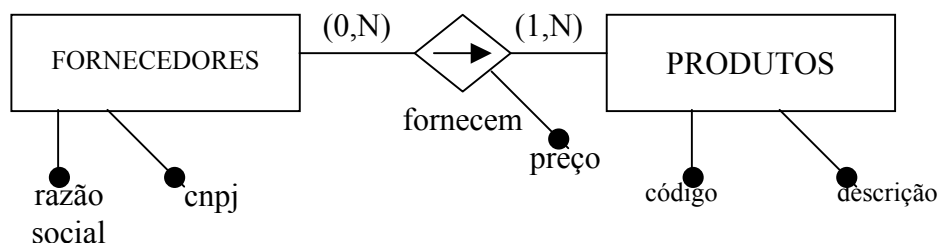


Figura 7.15 – Atributo de relacionamento.

Na prática, apenas os atributos de relacionamentos N:N são caracterizados como atributos de relacionamentos. No caso de relacionamentos 1:N ou N:1, os atributos do relacionamento podem ser perfeitamente caracterizados como atributos da entidade cuja a cardinalidade máxima é 1. No exemplo da figura 7.16, o atributo *data-de-lotação* pode ser perfeitamente caracterizado como atributo do conjunto de entidades FUNCIONÁRIOS.

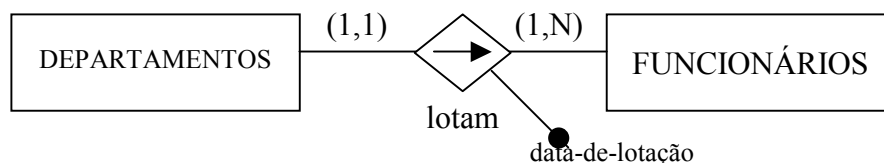


Figura 7.16 – Atributo de relacionamento caracterizado como atributo de uma das entidades.

Quando o relacionamento é N:N, há um teste que pode ser aplicado para se deduzir se um atributo é de um dos dois conjuntos de entidades ou se é do relacionamento. Seja o exemplo da figura 7.17:

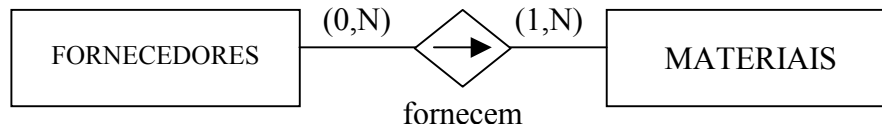


Figura 7.17 – Relacionamento N:N com atributos.

1. Fixa-se um material, por exemplo impressora, e varia-se os fornecedores deste material. Evidentemente o valor do atributo pode mudar. Por exemplo, a Casa do Analista vende uma impressora por R\$ 350, enquanto a loja Compute vende a mesma impressora por R\$ 310. Se o valor do atributo mudar ao variarmos o elemento do outro conjunto de entidades, é porque este não é atributo do primeiro conjunto de entidades, no caso MATERIAIS.
2. Procedimento análogo deve ser feito, agora, para a outra entidade. Fixando-se um fornecedor e variando-se os materiais temos: A Casa do Analista vende uma impressora por R\$ 350 e um microcomputador R\$ 1.700. Como o valor do atributo variou para a mesma entidade, é sinal de que ele não é atributo de FORNECEDORES.
3. Se não é atributo nem de MATERIAIS, nem de FORNECEDORES, então é um atributo do relacionamento entre os dois conjuntos de entidades, como mostra a figura 7.18.

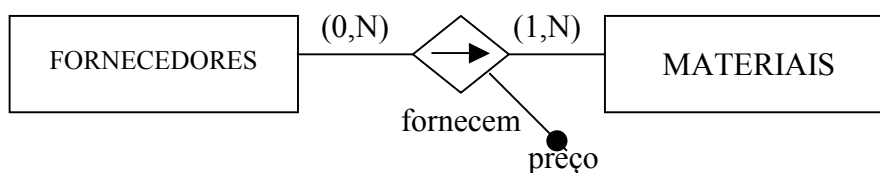


Figura 7.18 – Atributo do Relacionamento.

7.3.2 - Atributo de Entidade ou Nova Entidade ?

Pode não ser fácil decidir se um determinado elemento de informação deve ser tratado como atributo de uma entidade ou como uma segunda entidade relacionada à primeira. Analisemos o seguinte exemplo: Será que o código do departamento que oferece uma disciplina deve ser modelado como atributo de DISCIPLINAS, ou merece ser uma nova entidade relacionada à entidade DISCIPLINAS? De forma geral, convém tratar um elemento de informação como uma segunda entidade se:

- O elemento em questão tem atributos próprios;
- A segunda entidade resultante é relevante para a organização;
- O elemento em questão de fato identifica a segunda entidade;
- A segunda entidade pode ser relacionada a diversas ocorrências da entidade-1 (1:N);
- A segunda entidade relaciona-se a outras entidades que não a entidade-1.

Podemos notar que, no exemplo, todos os critérios anteriormente enumerados foram satisfeitos e, portanto, departamento deve ser tratado como uma nova entidade, como mostra a figura 7.19.



7.4.1 - Auto Relacionamento

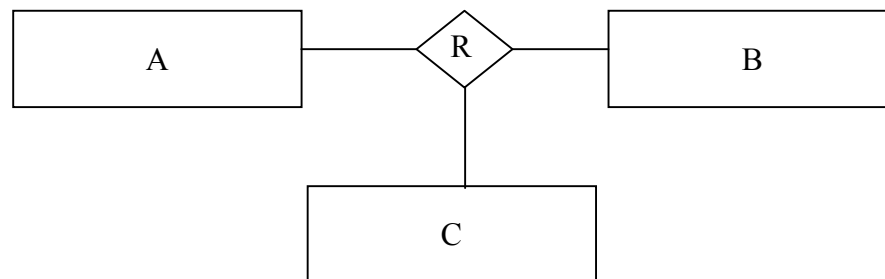
Diagrama de uma disciplina com seus pré-requisitos e pós-requisitos. Um retângulo centralizado contém o texto "DISCIPLINAS". À direita, um losango contém uma seta apontando para baixo. Duas linhas se conectam ao losango: uma superior rotulada "pré-requisito" e uma inferior rotulada "pós-requisito". Ambas as linhas terminam em parênteses contendo "0,N". À direita do losango, o texto "são pré-requisito para" indica a relação.

Neste caso, é necessário distinguir qual a atuação de cada elemento do conjunto de entidades no relacionamento e, portanto, é importante atribuir *papéis*. Assim no par (Cálculo I, Cálculo II), precisamos definir quem é pré-requisito de quem:

96

7.4.2 - Relacionamentos Múltiplos

São relacionamentos envolvendo mais de dois conjuntos de entidades. Um relacionamento ternário, por exemplo, só se caracteriza pelo terno, como mostra a figura 7.21. Os relacionamentos ternários normalmente são difíceis de se dar um nome e por isso é usual representá-los pelas iniciais das três entidades envolvidas, como mostra o exemplo da figura 7.22.



$$R \subseteq \{(a,b,c) / a \in A, b \in B, c \in C\}$$

Figura 7.21 – Relacionamento Ternário.

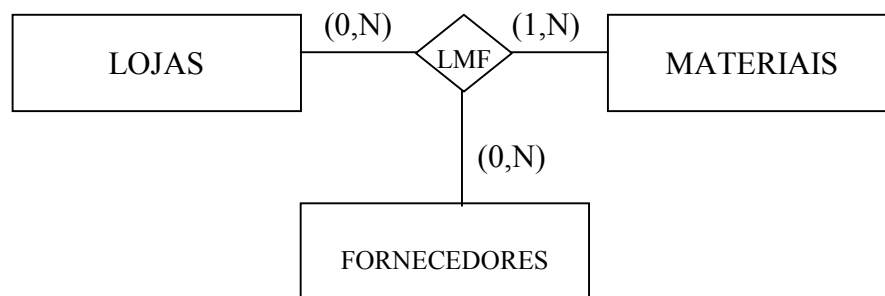
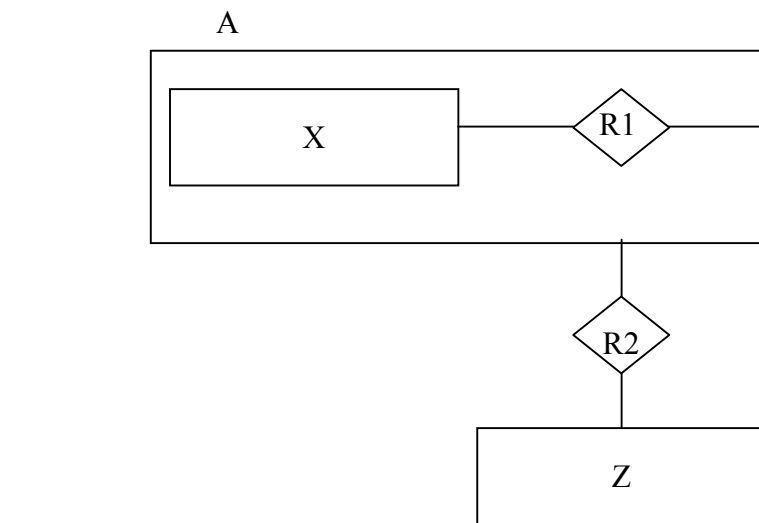


Figura 7.22 – Exemplo de Relacionamento Ternário.

7.4.3 - Agregações ou Agregados

Agregação é uma abstração através da qual relacionamentos entre duas entidades são tratados como entidades em um nível mais alto. Representa uma extensão do modelo originalmente proposto por Chen [Chen90], onde um relacionamento binário R e as entidades envolvidas X e Y são considerados uma única entidade A, agregando todas as informações. Esta “nova entidade”, a agregação, pode, então, relacionar-se com outras entidades do modelo, como mostra a figura 7.23.

A agregação é, normalmente, representada por um retângulo envolvendo as duas entidades e o relacionamento entre elas. A agregação deve ser dado um nome que represente esta “entidade de nível mais alto”, como mostra o exemplo da figura 7.24.



$$R1 \subseteq \{(x, y) / x \in X, y \in Y\}$$

$$R2 \subseteq \{((x, y), z) / (x, y) \in R1, z \in Z\}$$

Figura 7.23 – Agregação.

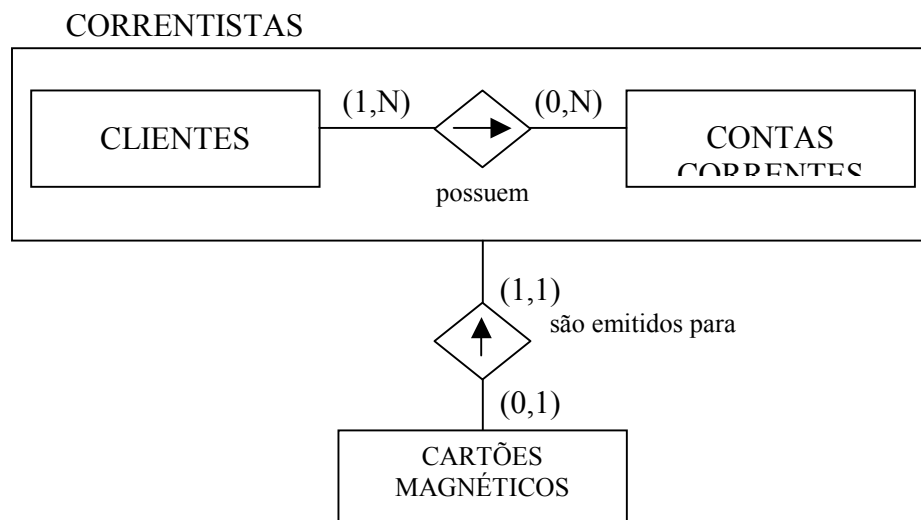


Figura 7.24 – Exemplo de Agregação.

Para prover maior facilidade na representação, muitas vezes, representaremos a agregação com um retângulo envolvendo apenas o relacionamento, como mostra o exemplo da figura 7.25.

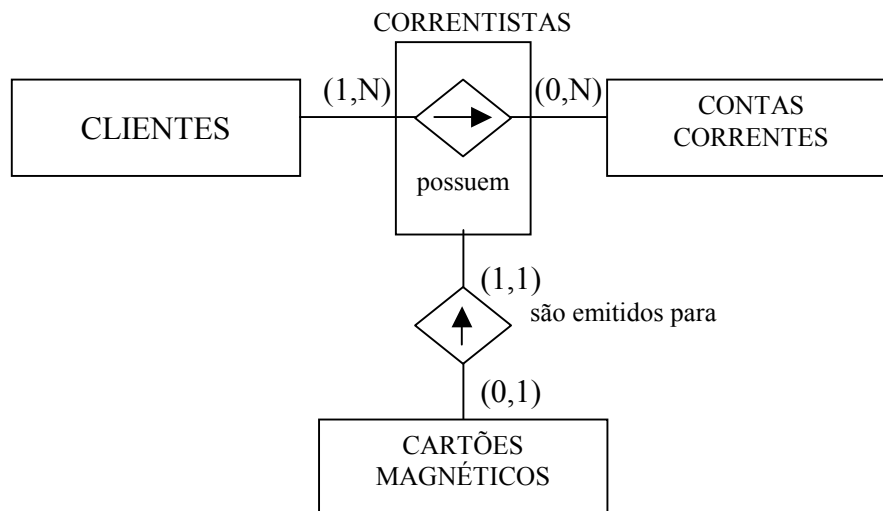


Figura 7.25 – Outra representação para Agregados.

É importante observar que agregações envolvendo relacionamentos $N:1$ ou $1:N$ não fazem sentido. Em relacionamentos desta natureza, cada entidade cuja cardinalidade máxima é 1 (Y) só aparece no máximo uma única vez no relacionamento e, conseqüentemente, já representa o par que eventualmente possa ocorrer. Assim, as duas versões apresentadas nas figuras 7.26 e 7.27 são equivalentes quanto às informações apresentadas e devemos utilizar sempre a versão da figura 7.27. Portanto, só devemos representar agregações de relacionamentos $N:N$.

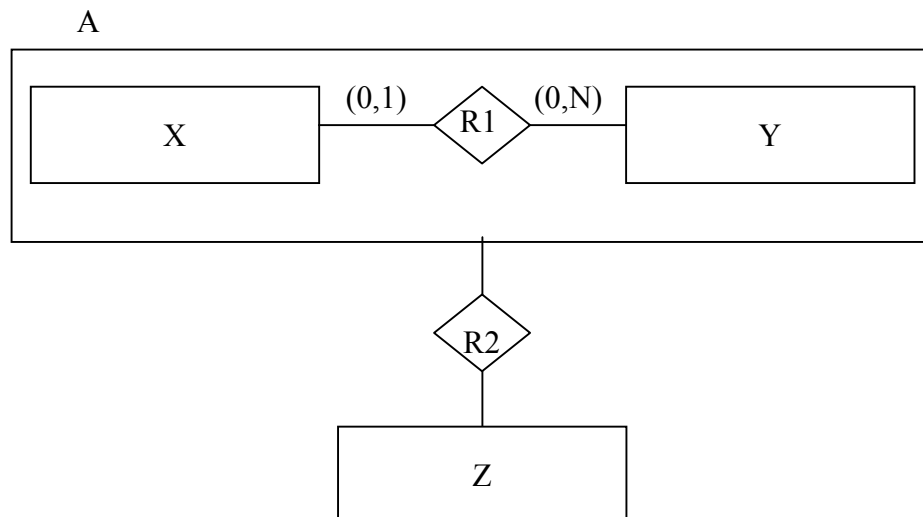


Figura 7.26 – Agregado em relacionamento $1:N$.

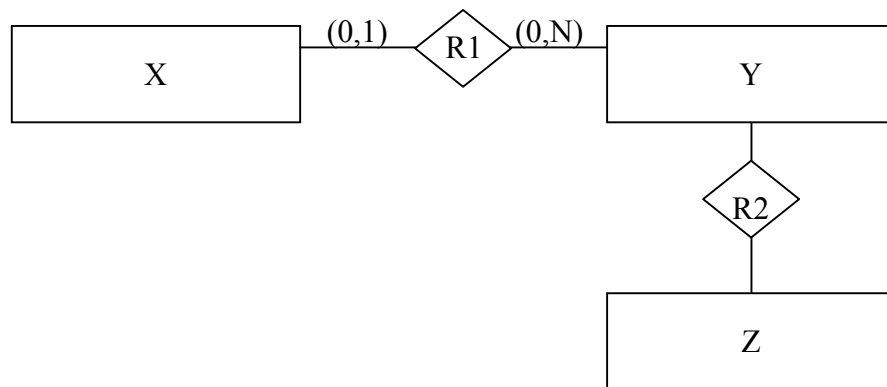


Figura 7.27 – Solução mais apropriada.

7.4.4 - Particionamentos de Conjuntos de Entidades

Muitas vezes, elementos de um conjunto de entidades do mundo real, se subdividem em categorias com atributos parcialmente distintos. Passa a ser interessante, então, representar os atributos comuns em um supertipo e os atributos distintos em subtipos. As abstrações que norteiam este procedimento são:

- Generalização: entidades de um nível mais baixo de abstração são agrupadas, dando origem a uma entidade de nível mais alto.
- Especialização: uma entidade de nível mais alto de abstração é desmembrada em várias entidades de nível mais baixo.

O particionamento pode ser:

- Disjunto: cada elemento do conjunto de entidades só se enquadra em uma das categorias.
- Não Disjunto: cada elemento pode ser enquadrado em mais de uma categoria.

A figura 7.28 mostra um exemplo de particionamento disjunto.

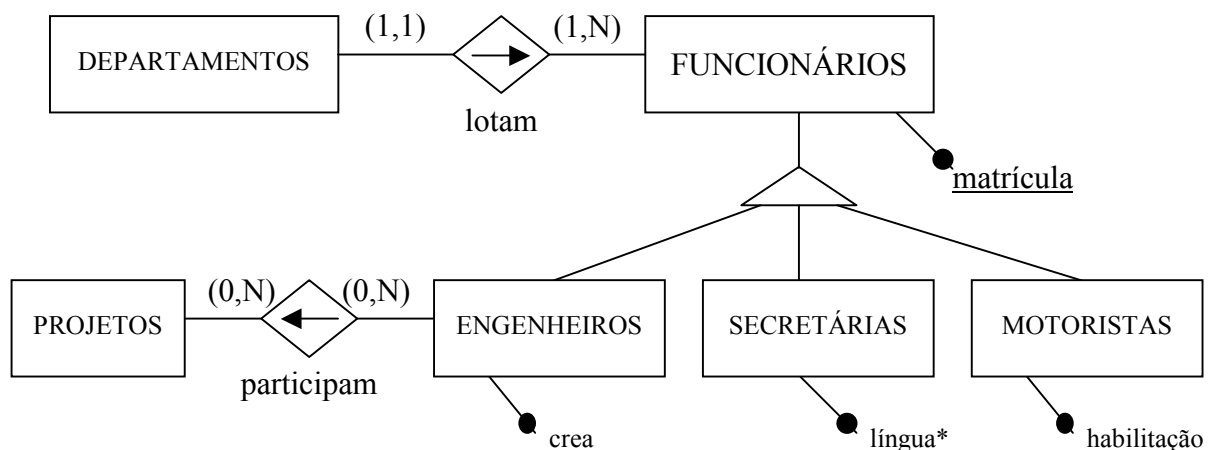


Figura 7.28 – Particionamento Disjunto de um Conjunto de Entidades.

7.4.5 - Conectores

São uma extensão do Modelo E/R para representar restrições de integridade no que diz respeito à totalidade dos relacionamentos. Podem ser de dois tipos:

- **Ou obrigatório:** Apenas um dos relacionamentos ocorre efetivamente, mas sempre um deles ocorre. No exemplo da figura 7.29, todo contrato é financiado (não existe contrato que não seja financiado), mas pode ser financiado ou por um banco ou por um fornecedor.

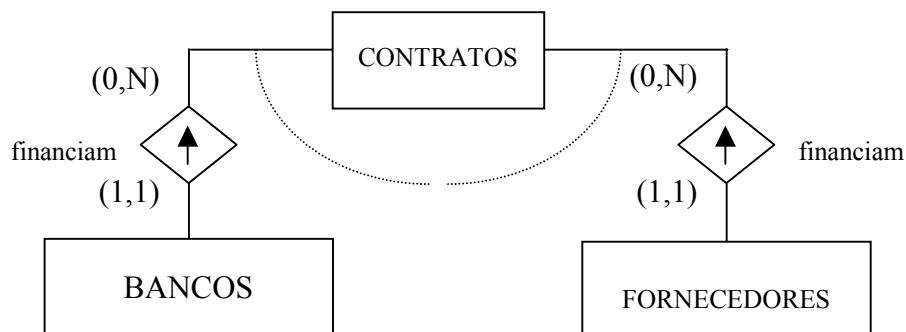


Figura 7.29 – Exemplo de Conector Ou-Obrigatório.

- **Ou opcional:** Apenas um dos relacionamentos ocorre efetivamente, mas pode ser que nenhum dos dois ocorra. No exemplo da figura 7.30, nem todo contrato é financiado. Mas se um contrato for financiado, ele será financiado ou por um banco ou por um fornecedor.

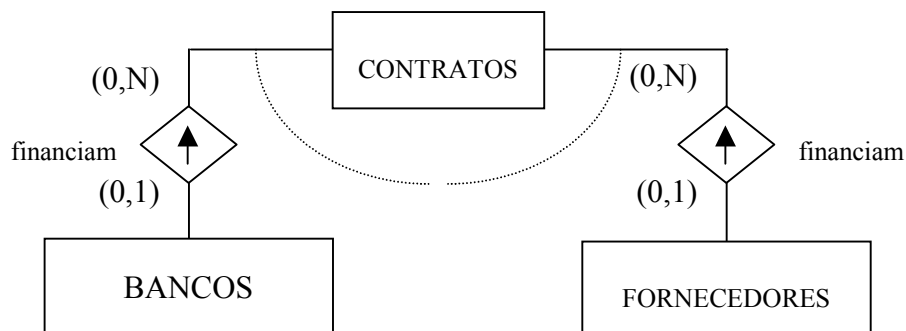


Figura 7.30 – Exemplo de Conector Ou-Opcional.

7.4.6 - Entidade Fraca

Existem entidades do mundo real que não têm existência própria, isto é, só aparecem no modelo quando relacionadas a outra entidade (dita forte), sendo seus atributos determinantes compostos por pelo menos dois campos, um deles um atributo da entidade forte. A figura 7.31 ilustra uma entidade fraca.

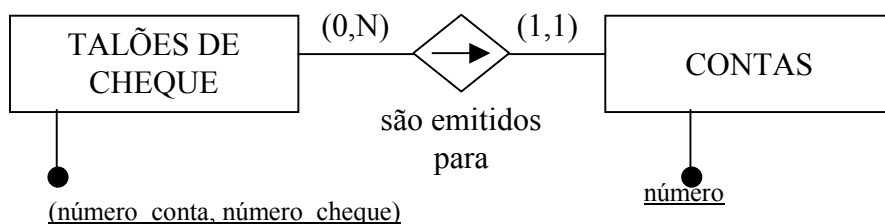


Figura 7.31 – Exemplo de Entidade Fraca.

7.5 – Dicionário de Dados

O Dicionário de Dados é uma listagem organizada de todos os elementos de dados pertinentes ao sistema, com definições precisas para que os usuários e desenvolvedores possam conhecer o significado de todos os itens de dados manipulados pelo sistema. Esta listagem contém, em ordem alfabética, as seguintes definições:

- entidades e relacionamentos com atributos de um DER.
- depósitos de dados e fluxos de dados dos DFDs, sendo que os primeiros devem corresponder às entidades e relacionamentos do DER.
- estruturas de dados que compõem os depósitos de dados ou fluxos de dados.
- elementos de dados que compõem os depósitos de dados, fluxos de dados ou estruturas de dados.

A figura 7.32 apresenta a notação adotada neste texto para elaboração de Dicionários de Dados.

Símbolo	Significado
=	é composto de
+	e
()	dado ou estrutura opcional
[]	dados ou estruturas alternativas (ou exclusivo)
n{ }m	repetição de dados ou estruturas, onde <i>n</i> representa o número mínimo de repetições e <i>m</i> o número máximo. Se <i>n</i> e <i>m</i> não são especificados, significa zero ou mais repetições.
* *	delimitadores de comentários
_____	atributo determinante

Figura 7.32 – Notação para Dicionários de Dados.

Os exemplos mostrados a seguir ilustram diversas situações e o emprego das notações.

(a) O cliente pode possuir um telefone.

CLIENTES = *clientes da livraria*

código-cliente + nome-cliente + endereço-cliente + (telefone-cliente)

(b) O cliente pode possuir um ou mais telefones.

CLIENTES = *clientes da livraria*

código-cliente + nome-cliente + endereço-cliente + {telefone-cliente}

(c) O cliente pode possuir até três telefones.

CLIENTES = *clientes da livraria*

código-cliente + nome-cliente + endereço-cliente + {telefone-cliente}3

(d) O cliente pode possuir telefone comercial, residencial ou ambos.

CLIENTES = *clientes da livraria*

código-cliente + nome-cliente + endereço-cliente + [telefone-comercial|
telefone-residencial | telefone-comercial + telefone-residencial]

Os nomes das entidades e relacionamentos (e, por conseguinte, dos depósitos de dados dos DFDs) deverão ser escritos no plural e com letras maiúsculas. Os atributos determinantes são assinalados por um sublinhado.

Referências Bibliográficas

- [Chen90] P. Chen. *Gerenciando Banco de Dados: A Abordagem Entidade-Relacionamento para Projeto Lógico*. McGraw-Hill, 1990.
- [Setzer87] W. Setzer. *Bancos de Dados*. 2ª Edição, Editora Edgard Blücher, 1987.
- [Yourdon90] E. Yourdon. *Análise Estruturada Moderna*. Editora Campus, 1990.

8. Modelagem Funcional

A partir deste momento, passaremos a nos preocupar com a modelagem das funções que o sistema deverá executar para atender aos anseios dos usuários do sistema.

A técnica mais difundida para esta finalidade é a utilização de Diagramas de Fluxo de Dados - DFDs, proposta por Gane e Sarson em [Gane83] e por De Marco em [De Marco83]. Muitos outros autores citam esta técnica em suas obras, sendo que destacamos como referência [Gane88] e [Yourdon90].

Um Diagrama de Fluxo de Dados, conforme proposto originalmente, é uma ferramenta *top-down* para modelagem de processos. Representa um sistema como uma rede de processos interligados entre si por fluxos de dados e depósitos de dados.

O DFD utiliza-se de quatro símbolos gráficos, visando representar os seguintes componentes: Processos, Fluxos de Dados, Depósitos de Dados e Entidades Externas. A figura 8.1 mostra a notação usada por Yourdon [Yourdon90], que será a adotada neste texto. Através da utilização desses quatro componentes, podemos representar satisfatoriamente os processos e interações entre os elementos de um sistema.

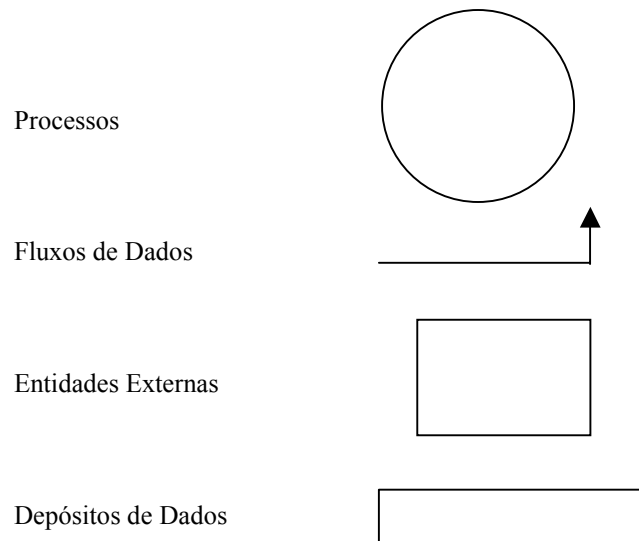


Figura 8.1 – Notação básica para construção de DFDs.

A idéia básica desta técnica é ir refinando o conhecimento das funções a partir da decomposição de um DFD em vários outros com níveis maiores de detalhamento, até se chegar a um entendimento completo das funções que o sistema deverá desempenhar.

Além dos Diagramas de Fluxo de Dados, são necessários para uma completa modelagem das funções:

- Dicionário de Dados;
- Descrição da lógica dos processos simples que não mereçam ser decompostos em outros.

Um DFD mostra as fronteiras do sistema: aquilo que não for uma Entidade Externa, será interno ao sistema, delimitando assim a abrangência do sistema. É uma ferramenta particularmente útil para a modelagem de sistemas por mostrar todas as relações entre dados (armazenados e que fluem no sistema) e os processos que manipulam e transformam esses dados.

Devemos lembrar que esta é uma técnica de modelagem conceitual e, portanto, não deve estar presa a nenhuma plataforma computacional de implementação.

8.1 - Conceitos Básicos

Processos

Representam as transformações e manipulações feitas sobre os dados em um sistema e correspondem a procedimentos ou funções que um sistema tem de prover. A ocorrência de um evento de um dos seguintes tipos deve ser representada como um processo em um DFD:

- transformações do conteúdo de um dado de entrada no conteúdo de um dado de saída, como mostra a figura 8.2;

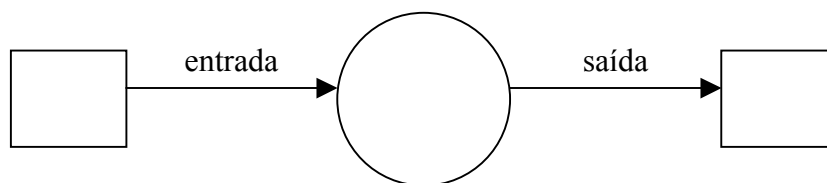


Figura 8.2 – Transformações de dados.

- inserções ou modificações do conteúdo de dados armazenados, a partir do conteúdo (possivelmente transformado) de dados de entrada, como mostra a figura 8.3;

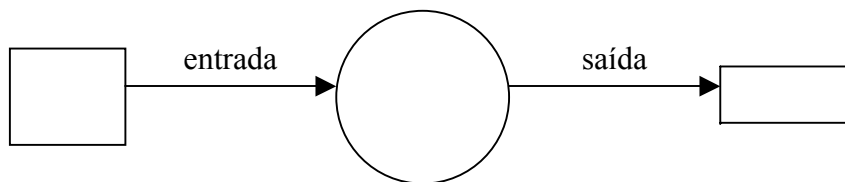


Figura 8.3 – Armazenamento de dados.

- transformações de dados previamente armazenados no conteúdo de um dado de saída, como mostra a figura 8.4.

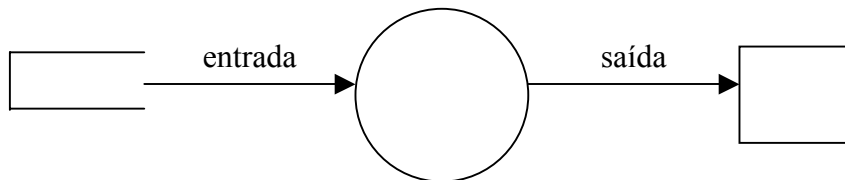


Figura 8.4 – Geração de dados de saída a partir de dados armazenados.

Um processo é representado por um círculo, com uma sentença simples (verbo + objetos) em seu interior e, opcionalmente, um identificador (número). A sentença deve tentar descrever o melhor possível a função a ser desempenhada, sem ambigüidades. Devem ser evitados nomes muito físicos (p. ex., gravar, imprimir, ...) ou muito técnicos (p. ex., apagar, fazer backup, ...) .

Os processos representados em um DFD não precisam ser necessariamente funções a serem informatizadas. Muitas vezes, para se prover um entendimento mais completo do sistema, processos manuais ou mistos (parte manual, parte informatizada) são representados.

Toda transformação de dados deve ser representada e, deste modo, não se admite ligação direta entre entidades externas e depósitos de dados. Por outro lado, devemos observar se um mesmo fluxo de dados entra e sai de um processo sem modificação, já que todo processo transforma dados.

Como já mencionado anteriormente, para uma completa modelagem das funções, são necessários, além dos DFDs, um Dicionário de Dados e as Especificações das Lógicas dos processos. Deste modo, só teremos um entendimento completo de um processo, após descrevermos sua lógica.

As especificações das lógicas dos processos só devem ser feitas para processos simples. Processos complexos devem ser decompostos em outros processos, até se atingir um nível de reduzida complexidade. Esta descrição não deve ser confundida com o detalhamento integral da lógica do processo que deverá ser feito na fase de projeto, mas deve servir de base para esta outra etapa.

Um processo só merece ser representado em um DFD quando a descrição de sua lógica utilizar aproximadamente uma página. Processos descritos em três ou quatro linhas são simples demais para serem tratados como processos em um DFD. Por outro lado, se a descrição da lógica do processo necessitar de três ou mais páginas, então esse processo está muito abrangente e não deve ser tratado como um único processo, mas sim ser decomposto em processos de menor complexidade. Para situações desta natureza, podemos utilizar duas técnicas : fissão ou explosão, como estudaremos a seguir.

Como regra geral, os fluxos de erro e exceção não devem ser mostrados nos diagramas, mas apenas na descrição da lógica dos processos. Esta regra só deve ser desrespeitada quando tais fluxos forem muito significativos para a comunidade usuária.

Fluxos de Dados

Fluxos de dados são utilizados para representar a movimentação de dados através do sistema. São simbolizados por setas, que identificam a direção do fluxo, e devem ter associado um nome o mais significativo possível, de modo a facilitar a validação do diagrama com os usuários. Esse nome deve ser um substantivo que facilite a identificação do dado (ou pacote de dados) transportado.

Um fluxo de dado em um DFD pode ser considerado como um caminho através do qual poderão passar uma ou mais estruturas de dados em tempo não especificado. Note que

em um DFD não se representam elementos de natureza não informacional, isto é, dinheiro, pessoas, materiais, etc...

Devemos observar se um fluxo de dados entra e sai de um processo sem modificação. Isto representa uma falha, haja visto que um processo transforma dados. Embora possa parecer um tanto óbvio, é bom lembrar que um mesmo conteúdo pode ter diferentes significados em pontos distintos do sistema e, portanto, os fluxos devem ter nomes diferentes. No DFD da figura 8.5, um mesmo conjunto de informações sobre um cliente tem significados diferentes quando passa pelos fluxos *dados-novo-cliente* e *dados-cliente*. No primeiro caso, os dados ainda não foram validados e, portanto, podem ser válidos ou inválidos, enquanto, no segundo fluxo, esses mesmos dados já foram validados.

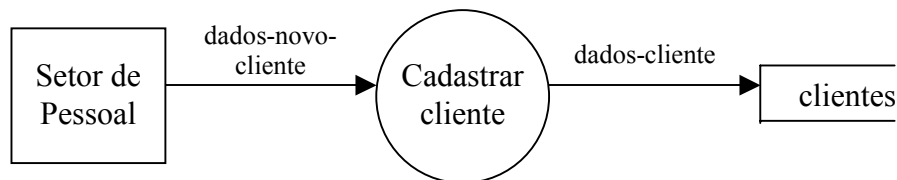


Figura 8.5 – Mesmo conteúdo de dados em fluxos diferentes.

Fluxos de dados que transportam exatamente o mesmo conteúdo de/para um depósito de dados, não precisam ser nomeados. No exemplo da figura 8.5, se o fluxo *dados-cliente* apresentar exatamente o mesmo conteúdo do depósito *clientes*, não há necessidade de nomeá-lo, como mostra a figura 8.6.

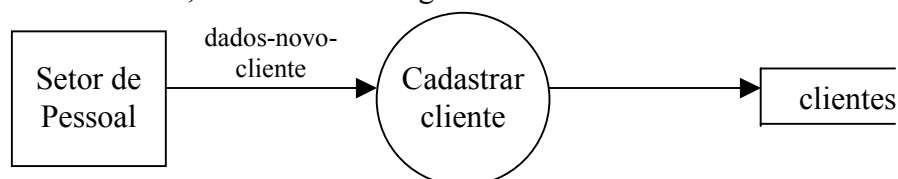


Figura 8.6 – Fluxo de dados não nomeado.

Fluxos de erro ou exceção (no exemplo, *dados-cliente-inválidos*) só devem ser mostrados em um DFD, se forem muito significativos para o seu entendimento. Caso contrário, devem ser tratados apenas na descrição da lógica do processo.

Setas ramificadas significam que o mesmo fluxo de dados está indo de uma fonte para dois destinos diferentes, isto é, cópias do mesmo pacote de dados estão sendo enviadas para diferentes partes do sistema, como mostra a figura 8.7.

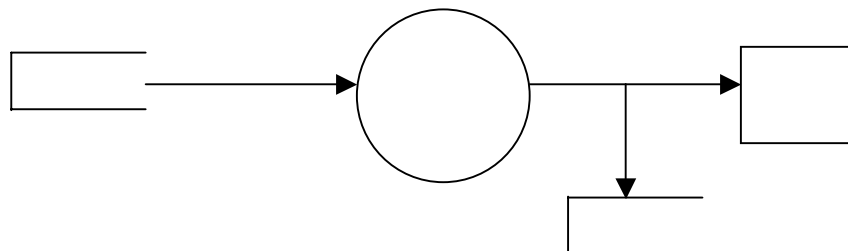


Figura 8.7 – Fluxo ramificado.

Quando for necessário cruzar fluxos de dados em um DFD, devemos utilizar um arco, como mostra a figura 8.8.

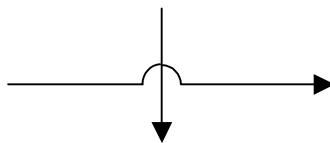


Figura 8.8 – Fluxos de dados que se cruzam em um diagrama.

É importante realçar que DFDs não indicam a seqüência na qual fluxos de dados entram ou saem de um processo.

Depósitos de Dados

Depósitos de dados são pontos de retenção permanente ou temporária de dados, que permitem a integração entre processos assíncronos, isto é, processos realizados em tempos distintos. Sem nos comprometermos quanto ao aspecto físico, representam um local de armazenamento de dados entre processos.

Um depósito de dados é representado por um retângulo sem a linha lateral direita, com um nome e um identificador (opcional) em seu interior. Às vezes, para evitar o cruzamento de linhas de fluxos de dados ou para impedir que longas linhas de fluxos de dados saiam de um lado para outro do diagrama, um mesmo depósito de dados pode ser representado mais de uma vez no diagrama. Nesta situação, adicionamos uma linha vertical na lateral esquerda do retângulo, como mostra a figura 8.9.



Figura 8.9 – Notação para depósitos de dados.

Um depósito de dados não se altera quando um pacote de informação sai dele através de um fluxo de dados. Por outro lado, um fluxo para um depósito representa uma das seguintes ações:

- uma inclusão, isto é, um ou mais novos pacotes de informação estão sendo introduzidos no depósito;
- uma atualização, ou seja, um ou mais pacotes estão sendo modificados, sendo que isso pode envolver a alteração de todo um pacote, ou apenas de parte dele;
- uma exclusão, isto é, pacotes de informação estão sendo removidos do depósito.

A semântica dos acessos aos depósitos de dados é mostrada na figura 8.10.

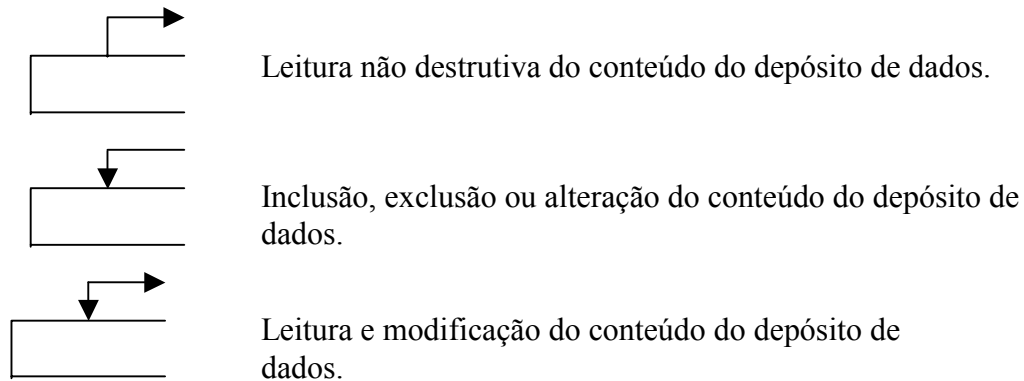


Figura 8.10 – Semântica dos acessos a depósitos de dados em um DFD.

Quando examinamos fluxos de dados que entram ou saem de um depósito, surge uma dúvida: o fluxo representa um único pacote, múltiplos pacotes, partes de um pacote, ou partes de vários pacotes de dados? Em algumas situações, essas dúvidas podem ser respondidas pelo simples exame do rótulo do fluxo e, para tal, adotamos a seguinte convenção:

- se um fluxo não possuir rótulo ou tiver o mesmo rótulo do depósito de dados, então um pacote inteiro de informação ou múltiplas instâncias do pacote inteiro estão trafegando pelo fluxo;
- se o rótulo de um fluxo nomeado for diferente do rótulo do depósito, então as informações que estão trafegando são um ou mais componentes (partes) de um ou mais pacotes, e estarão definidas no dicionário de dados.

Muitas vezes, diferentes sistemas compartilham uma mesma base de dados e, portanto, vários sistemas poderão estar lendo e atualizando os conteúdos de um mesmo depósito de dados. É interessante mostrar este fato explicitamente no DFD e, neste caso, podemos notar três situações distintas:

- O sistema em questão apenas lê as informações do depósito de dados, não sendo responsável por qualquer alteração de seu conteúdo. Neste caso, utilizamos a notação da figura 8.11.

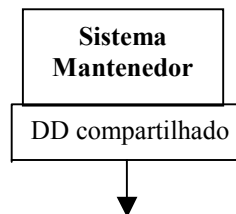


Figura 8.11 – Sistema apenas acessando depósito de dados mantido por outro sistema.

- O sistema em questão apenas gera as informações que são utilizadas por outros sistemas. Representaremos esta situação segundo a notação da figura 8.12.

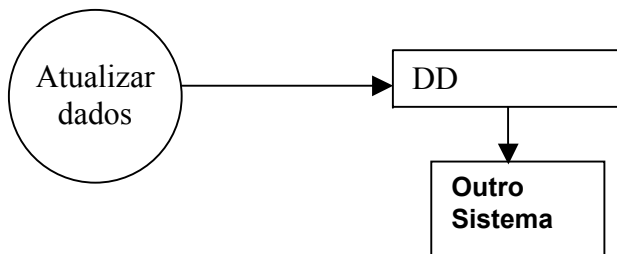


Figura 8.12 – Sistema atualizando dados utilizados por outro sistema.

- Ambos os sistemas atualizam o depósito de dados. A notação para esta situação é mostrada na figura 8.13.

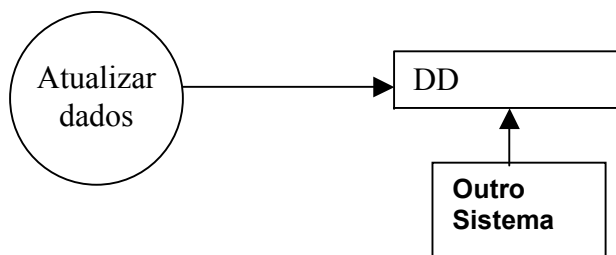


Figura 8.13 – Ambos os sistemas atualizando dados de um mesmo depósito.

Estas notações são exceções à regra de que os dados não devem fluir diretamente entre uma entidade externa e um depósito de dados, sem passar por um processo responsável pela transferência dos dados. A fora as situações anteriormente descritas, devemos observar a integridade de um depósito de dados segundo dois prismas:

- Observar se todos os elementos de dados que fazem parte do depósito têm como efetivamente chegar lá, isto é, fazem parte de pelo menos um fluxo de dados que chega ao depósito.
- Observar se todos os elementos de dados que fazem parte do depósito são, em algum momento, solicitados por um processo, isto é, fazem parte de pelo menos um fluxo de dados que sai do depósito.

Entidades Externas

Entidades externas ou terminadores são fontes ou destinos de dados do sistema. Representam os elementos do ambiente com os quais o sistema se comunica. Tipicamente, uma entidade externa é uma pessoa (p.ex. um cliente), um grupo de pessoas (p. ex. um departamento da empresa ou outras instituições) ou um outro sistema que interaja com o sistema em questão. Uma entidade externa deve ser identificada por um nome e

representada por um retângulo. Assim como no caso dos depósitos de dados, em diagramas complexos, podemos desenhar um mesmo terminador mais de uma vez, para se evitar o cruzamento de linhas de fluxos de dados ou para impedir que longas linhas de fluxos de dados saiam de um lado a outro do diagrama. Neste caso, convencionou-se utilizar um traço diagonal no canto inferior direito do símbolo da entidade externa, como mostra a figura 8.14..



Figura 8.14 – Notações para representar entidades externas.

Ao identificarmos alguma coisa ou sistema como uma entidade externa, estamos afirmando que esta entidade está fora dos limites do sistema em questão e, portanto, fora do controle do sistema que está sendo modelado. Assim, qualquer relacionamento existente entre entidades externas não deve ser mostrado em um DFD.

Se percebermos que, em algum ponto do sistema, descrevemos algo que ocorre dentro de uma entidade externa ou relacionamentos entre entidades externas, é necessário reconhecer que a fronteira do sistema é na realidade mais ampla do que foi estabelecido inicialmente e, portanto, deve ser revista.

Uma vez que os terminadores são externos ao sistema, os fluxos de dados que os interligam aos diversos processos representam a interface entre o sistema e o mundo externo.

8.2 - Construindo DFDs

Como já mencionado no estudo sobre processos, é uma boa prática manter um certo nível de complexidade nos processos representados em um DFD. Esse nível de complexidade pode ser estabelecido pelo tamanho da especificação da lógica do processo ou pelo número de processos em um diagrama. Se tal nível de complexidade for superado, devemos utilizar uma das seguintes técnicas para decompor o DFD: fissão ou explosão.

Fissão

Na fissão, o processo complexo deve ser substituído no próprio DFD do sistema por um número de processos mais simples. Por exemplo, se um processo requer 8 páginas de especificação de lógica, ele pode ser substituído por 4 processos, cada um deles tendo aproximadamente 2 páginas.

O problema na utilização desta técnica é a sobrecarga a que o diagrama poderá ficar sujeito, dificultando sua leitura.

Explosão

O processo original permanece no diagrama, sendo criado um novo DFD de nível inferior, consistindo de processos menos complexos. Assim, um projeto não é representado por um único DFD, mas sim por um conjunto de DFDs em vários níveis de decomposição funcional.

Quando a explosão é utilizada, alguns aspectos importantes devem ser observados. O primeiro deles diz respeito ao número de níveis que devem ser esperados em um sistema. A priori, este número não deve ser pré-fixado, mas lembre-se que o número total de processos cresce exponencialmente quando se passa de um nível para o imediatamente inferior. De uma maneira geral, quando a explosão é utilizada, pelo menos três níveis são especificados, apesar de algumas vezes só serem necessários dois.

Tipicamente são quatro os níveis de representação:

- **C – Contexto:** mostra o sistema como uma “caixa-preta”, trocando informações (fluxos de dados) com entidades externas ao sistema. Define o escopo de abrangência do sistema, indicando que se está renunciando à possibilidade de examinar qualquer coisa além da fronteira definida pelas entidades externas.
- **0 (Zero) – Geral ou de Sistema:** é uma decomposição do diagrama de contexto, mostrando o funcionamento do sistema em questão, isto é, as grandes funções do sistema e as interfaces entre elas. Os processos nesse diagrama recebem os números 1, 2, 3, etc... É necessário assegurar a coerência entre os diagramas **C** e **0**, isto é, assegurar que os fluxos de dados entrando ou saindo do diagrama **0** efetivamente reproduzem as entradas e saídas do diagrama **C**. Neste diagrama, devem aparecer os depósitos de dados necessários para a sincronização dos processos.
- **N – Detalhe:** Uma diagrama de detalhe representa a decomposição de um dos processos do diagrama **0** e, portanto, é nomeado com o número e o nome do processo que está sendo detalhado. A princípio, deverão ser elaborados diagramas **N** para os processos do diagrama **0** que sejam complexos e, portanto, careçam de decomposição. É necessário resguardar a coerência entre o diagrama **0** e cada diagrama detalhado elaborado. Os processos em um diagrama **N** são numerados com o número do processo que está sendo detalhado (p. ex., 2) e um número seqüencial, separados por um ponto (p. ex., 2.1, 2.2, etc.).
- **E – Detalhe Expandido:** um diagrama deste tipo representa a decomposição de um dos processos do diagrama **N**. Este nível de decomposição pode vir a ser necessário caso um processo do nível **N** seja ainda muito complexo. Este nível pode ser desdobrado sucessivamente até se atingir o grau necessário de simplicidade. Entretanto, se muitos níveis forem necessários, cuidado! Provavelmente, o contexto funcional da aplicação (diagrama de contexto) está muito abrangente e merece revisão.

Fissão ou Explosão?

Recomenda-se o uso da fissão para sistemas de pequeno a médio porte, em que a leitura do diagrama não fica prejudicada pelo aparecimento de mais alguns processos no diagrama de sistema. A fissão possui a vantagem de representar todo o sistema em um único DFD, não sendo necessário recorrer a outros diagramas para se obter um entendimento completo de suas funções. Em sistemas maiores, o uso da fissão pode se tornar inviável, sendo recomendado, então, o uso da explosão.

Recomendações para a Construção de DFDs

1. Escolha nomes significativos para todos os elementos de um DFD. Utilize termos empregados pelos usuários no domínio da aplicação.
2. Os processos devem ser numerados de acordo com o diagrama a que pertencem.
3. Evite desenhar DFDs complexos.
4. Cuidado com os processos sem fluxos de dados de entrada ou de saída.
5. Cuidado com os depósitos de dados que só possuem fluxos de dados de entrada ou de saída.
6. Depósitos de dados permanentes devem manter estreita relação com os conjuntos de entidades e de relacionamentos do modelo ER.
7. Fique atento ao princípio de conservação de dados, isto é, dados que saem de um depósito devem ter sido previamente lá colocados e dados produzidos por um processo têm de ser passíveis de serem gerados por esse processo.
8. Quando do uso de explosão, os fluxos de dados que entram e saem em um diagrama de nível superior devem entrar e sair no nível inferior que o detalha.
9. Mostre um depósito de dados no nível mais alto em que ele faz a sincronização entre dois ou mais processos. Passe a representá-lo em todos os níveis inferiores que detalham os processos envolvidos.
10. Não represente no DFD fluxos de controle ou de material. Como o nome indica, DFDs representam fluxos de dados.
11. Só especifique a lógica de processos primitivos, ou seja, aqueles que não são detalhados em outros diagramas.

8.3 - Técnicas de Especificação de Processos

Quando chegamos a um nível de especificação em que os processos não são mais decomponíveis, precisamos complementar esta especificação com descrições das lógicas desses processos. A especificação de processos deve ser feita de forma que possa ser validada por analistas e usuários. Entretanto, encontramos muitos problemas na descrição de forma narrativa, entre os quais podemos citar:

- Uso de expressões do tipo: mas, todavia, a menos que ...
Por exemplo, qual a diferença entre as declarações abaixo ?
 - Somar A e B, a menos que A seja menor que B, onde, neste caso, subtrair A de B.
 - Somar A e B. Entretanto, se A for menor que B, a resposta será a diferença entre B e A.
 - Somar A e B, mas subtrair A de B quando A for menor que B.
 - total é a soma de B e A. Somente quando A for menor que B é que a diferença deve ser usada como o total.

Ao analisarmos estas frases, notamos que não existe diferença lógica entre elas, entretanto as formas narrativas apresentadas mascaram a semelhança existente. Se ao invés de usarmos uma forma narrativa, usarmos uma forma padrão do tipo *se-então-senão*, teremos maior clareza e validação.

se A < B
então TOTAL \leftarrow B - A;
senão TOTAL \leftarrow A + B;
fim-se;

- Uso de comparativos como: Maior que / Menor que, Mais de / Menos de.
Seja a seguinte declaração: “Até 20 unidades, sem desconto. Mais de R\$20, 5% de desconto.”
E exatamente 20 unidades, que tratamento deve ser dado ?
- Ambigüidades do E/OU.
Seja a seguinte declaração: “Clientes que gerarem mais de um milhão de cruzeiros em negócios por ano e possuírem um bom histórico de pagamentos **ou** que estiverem conosco há mais de 20 anos, devem receber tratamento prioritário.”
Quem deverá receber tratamento prioritário? Clientes com mais de 1 milhão em negócios por ano que possuírem bom histórico de pagamentos? Clientes com mais de 20 anos? Clientes com mais de 1 milhão e (ou bom histórico, ou mais de 20 anos)?
Note que pela declaração não fica claro quando deverá ser aplicado o tratamento prioritário.
- Uso de Adjetivos Indefinidos
Na declaração do item anterior, o que é um **bom** histórico de pagamentos? Devemos tomar cuidado ao utilizarmos adjetivos indefinidos e quando o fizermos, tomarmos o cuidado de defini-los.

Para administrar os problemas oriundos da narrativa, são utilizadas técnicas de especificação de processos, entre as quais podemos citar:

- Português Estruturado
- Tabelas de Decisão

- Árvores de Decisão
- Combinação das técnicas acima

8.3.1 - Português Estruturado

O Português Estruturado é um subconjunto do Português, cujas sentenças são organizadas segundo as três estruturas de controle introduzidas pela Programação Estruturada: seqüência, seleção e repetição.

- **Instruções de Seqüência:** grupo de instruções a serem executadas que não tenham repetição e não sejam oriundas de processos de decisão. São escritas na forma imperativa, como no exemplo abaixo.

```
obter ...  
atribuir ...  
armazenar ...
```

- **Instruções de Seleção:** quando uma decisão deve ser tomada para que uma ação seja executada, utilizamos uma instrução de seleção. As instruções de seleção são expressas como uma combinação *se-então-senão*, conforme abaixo.

```
se    <condição>  
    então grupo_de_ações_1;  
    então grupo_de_ações_2;  
fim-se;
```

Exemplo:

```
se  Número_de_Dependentes = 0  
    então Salário_Família = 0;  
    então Salário_Família = Salário_Mínimo / 3;  
fim-se;
```

Quando existirem várias ações dependentes de uma mesma condição, que sejam mutuamente exclusivas, podemos utilizar uma estrutura do tipo *caso*, conforme abaixo.

```
caso <condição> =  
    valor_1 : grupo_de_ações_1;  
    valor_2 : grupo_de_ações_2;  
    ...    ...  
    valor_n : grupo_de_ações-N;  
fim-caso;
```

Exemplo:

```
caso opção =  
    1 :  incluir novo cliente;  
    2 :  excluir cliente existente;  
    3 :  alterar dados de cliente;  
    então :  executar rotina de erro;  
fim-caso;
```

- **Instruções de Repetição:** Aplicadas quando devemos executar uma instrução, ou um grupo de instruções, repetidas vezes. A estrutura de repetição pode ser usada de três formas distintas:

1. **para cada “X” faça**
 grupo_de_ações;
 fim-para;

Exemplo:

```
para cada Aluno faça
    Média = (Prova_1 + Prova_2) / 2;
    imprima Média;
fim-para;
```

2. **enquanto <condição for verdadeira> faça**
 grupo_de_ações;

fim-enquanto;

Exemplo:

```
enquanto existir registro faça
    ler registro;
    consistir dados;
fim-enquanto;
```

3. **repita**
 grupo_de_ações;
 até que <condição seja verdadeira>;

Exemplo:

```
repita
    ler registro
    consistir dados
até que todos os registros do arquivo tenham sido processados;
```

Uma especificação de processo em Português Estruturado deve possuir as seguintes características gerais:

- deve ser clara, concisa, completa e livre de ambigüidades;
- todos os dados citados na especificação que estejam definidos no dicionário de dados devem ser sublinhados;
- os dados definidos localmente não são sublinhados;
- os depósitos de dados, além de sublinhados, devem ser escritos com letras maiúsculas;

- sempre que um comando de seleção ou repetição for utilizado, os comandos do bloco interno (grupo_de_ações) devem estar identados, de modo a dar a clareza de que esses comandos fazem parte das ações da seleção ou repetição.

8.3.2 – Árvore de Decisão

Árvores de Decisão são excelentes para mostrar a estrutura de decisão de um processo. Os ramos da árvore correspondem a cada uma das possibilidades lógicas. É uma excelente ferramenta para esquematizar a estrutura lógica e para obter do usuário a confirmação de que a lógica expressada está correta. De forma clara e objetiva, permite a leitura da combinação das circunstâncias que levam a cada ação.

Como podemos notar pelo exemplo da figura 8.15, uma Árvore de Decisão é muito boa para representar a lógica decisória. Entretanto, se for necessário descrever a lógica de um processo como um conjunto de instruções, combinando decisões e ações intermediárias, a árvore de decisão deve ser preterida em favor do português estruturado ou combinada a ele.

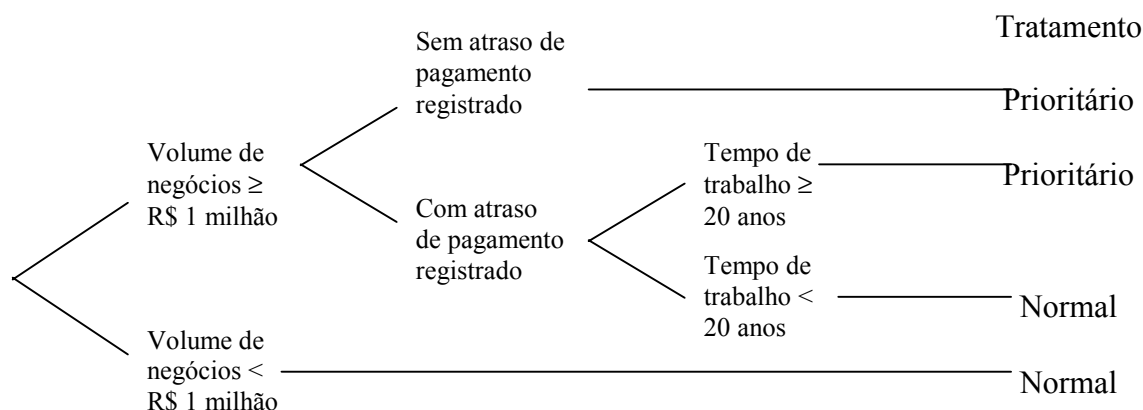


Figura 8.15 – Exemplo de Árvore de Decisão.

8.3.3 – Tabelas de Decisão

Tabelas de decisão são usadas em aplicações semelhantes às das árvores de decisão. As árvores de decisão são mais indicadas, quando o número de decisões for pequeno e nem todas as combinações de condições forem possíveis. As tabelas de decisão aplicam-se melhor a situações em que o número de ações é grande e ocorrem muitas combinações de condições. Também devemos utilizar tabelas de decisão se existirem dúvidas de que a árvore de decisão não mostra toda a complexidade do problema.

O formato básico de uma tabela de decisão é mostrado na figura 8.16.

Nome da Tabela	
Condições	Combinações
Ações	Regras

Figura 8.16 – Formato básico de uma Tabela de Decisão.

A construção de uma tabela de decisão envolve os seguintes passos:

1. Levantar as ações do processo;
2. Identificar as condições que determinam estas ações;
3. Identificar os estados possíveis de cada condição;
4. Identificar as combinações dos estados das condições;
5. Construir uma coluna para cada combinação de condições;
6. Preencher cada coluna com as regras das ações correspondentes;
7. Verificar se o entendimento foi correto;
8. Alterar a tabela até obter total concordância dos usuários;
9. Se possível, compactar a tabela.

Em função do tipo das condições, temos dois tipos de tabelas:

- Tabela de Entrada Limitada: os valores de uma condição se limitam a dois. Exemplos típicos deste tipo de tabelas são as tabelas cujas condições são escritas sob a forma de perguntas, de modo que as respostas sejam “sim” ou “não”, como mostra o exemplo da figura 8.17.

Tratamento de Clientes								
Volume de Negócios \geq R\$ 1 milhão?	S	S	S	S	N	N	N	N
Atraso de pagamento registrado?	N	N	S	S	N	N	S	S
Tempo de trabalho \geq 20 anos?	S	N	S	N	S	N	S	N
Tratamento Prioritário	X	X	X					
Tratamento Normal				X	X	X	X	X

Figura 8.17 – Tabela de Entrada Limitada.

- Tabela de Entrada Ampliada: Uma condição pode ter mais de dois estados diferentes, como no exemplo da figura 8.18.

Cobrança de Fretes												
Meio de Transporte	F	F	F	F	R	R	R	R	M	M	M	M
Tipo de Entrega	R	R	N	N	R	R	N	N	R	R	N	N
Peso	L	P	L	P	L	P	L	P	L	P	L	P
R\$ 100/Kg					X				X			
R\$ 50/Kg	X					X	X			X	X	
R\$ 10/Kg		X	X	X				X				X

Meio de Transporte: Ferroviário (F), Rodoviário (R), Marítimo (M).

Tipo de Entrega: Rápida (R) – até 5 dias úteis; Normal (N) – até 30 dias.

Peso: Leve (L): $\leq 100\text{kg}$; Pesado (P): $> 100\text{Kg}$

Figura 8.18 – Tabela de Entrada Ampliada.

Muitas vezes, grupos de condições levam à mesma ação. Para estes casos, podemos utilizar tabelas compactadas, como a do exemplo 8.19.

Tratamento de Clientes				
Volume de Negócios \geq R\$ 1 milhão?	S	S	S	N
Atraso de pagamento registrado?	N	S	S	-
Tempo de trabalho \geq 20 anos?	-	S	N	-
Tratamento Prioritário	X	X		
Tratamento Normal			X	X

Figura 8.19 – Tabela Compactada.

Quando uma única tabela se tornar muito grande ou complexa, podemos utilizar tabelas encadeadas, onde uma tabela faz referência a outra, como mostra o exemplo da figura 8.20.

Tratamento de Clientes - 1		
Volume de Negócios \geq R\$ 1 milhão?	S	N
Tratamento de Clientes - 2	X	
Tratamento Normal		X

Tratamento de Clientes - 2				
Atraso de pagamento registrado?	N	N	S	S
Tempo de trabalho \geq 20 anos?	S	N	S	N
Tratamento Prioritário	X	X	X	
Tratamento Normal				X

Figura 8.20 – Tabelas Encadeadas.

Referências Bibliográficas

- [De Marco83] T. De Marco. *Análise Estruturada e Especificação de Sistemas*. Editora Campus, 1983.
- [Gane83] C. Gane, T. Sarson. *Análise Estruturada de Sistemas*. Livros Técnicos e Científicos Editora, 1983.
- [Gane88] C. Gane. *Desenvolvimento Rápido de Sistemas*. Livros Técnicos e Científicos Editora, 1988.
- [Yourdon90] E. Yourdon. *Análise Estruturada Moderna*. Editora Campus, 1990.