

UFES - Universidade Federal do Espírito Santo

Projeto de Sistemas

Notas de Aula

Ricardo de Almeida Falbo

E-mail: falbo@inf.ufes.br

2003/1

Índice

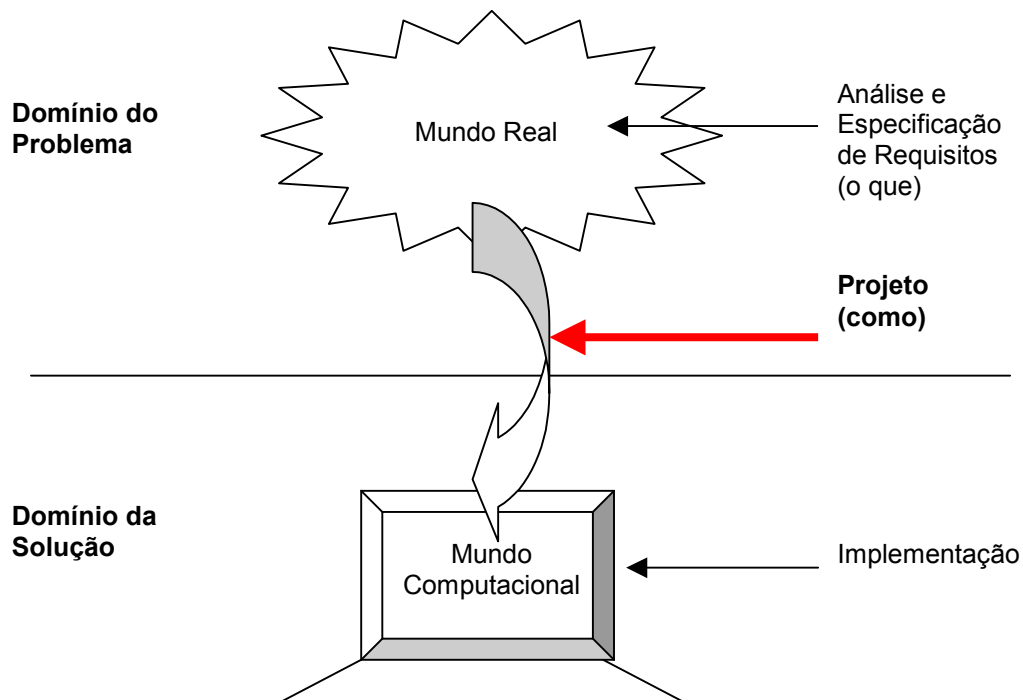
Capítulo 1 - Introdução	1
1.1 – A Fase de Projeto	1
1.2 – Conceitos Básicos de Projeto	2
1.3 – Princípios de Projeto	3
1.4 – Qualidade do Projeto	3
1.5 – Tecnologia Imperfeita e Requisitos Tecnológicos	6
Capítulo 2 – Projeto Arquitetural	8
2.1 – Objetivos do Projeto Arquitetural	8
2.2 – A Arquitetura Cliente-Servidor	9
2.3 – Coletando Estatísticas e Restrições	13
2.4 – Estratégias de Acesso Oportuno a Dados	18
2.5 – Sumário da Determinação da Distribuição Geográfica	20
2.6 – Segurança	20
2.7 – Modelo de Tarefas	21
2.8 – Atividades Tecnológicas	22
Capítulo 3 – Projeto de Interface com o Usuário	23
3.1 – Aspectos Gerais de Interface com o Usuário	24
Capítulo 4 – Projeto de Bancos de Dados Relacionais	27
4.1 – O Modelo Relacional	27
4.2 – Diagrama Relacional	30
Capítulo 5 - Projeto Orientado a Objetos	32
5.1 – Projeto da Arquitetura do Sistema	33
5.2 – Projeto da Componente do Domínio do Problema	36
5.3 – Projeto da Componente de Interface com o Usuário	36
5.4 – Projeto da Componente de Gerência de Tarefas	37
5.5 – Projeto da Componente de Gerência de Dados	39
5.6 – Projeto de Objetos	42
5.7 – Critérios de Qualidade de Projeto OO	43
5.8 – Revisão do Documento de Projeto	44
5.9 – Padrões de Projeto (<i>Design Patterns</i>)	45
Capítulo 6 – Projeto Estruturado de Sistemas	62
6.1 – Projeto de Dados	62
6.2 – Projeto Modular de Programas	69

1. Introdução

Referências: Cap. 13 (Pressman, 2002), Caps. 2 e 3 (Xavier et al., 1995)

O projeto de software encontra-se no núcleo técnico do processo de desenvolvimento de software e é aplicado independentemente do modelo de ciclo de vida e paradigma adotados. É iniciado assim que os requisitos do software tiverem sido modelados e especificados, correspondendo à primeira dentre as três atividades técnicas – projeto, implementação e testes – requeridas para se construir e verificar um sistema de software.

Enquanto a fase de análise pressupõe que dispomos de tecnologia perfeita (capacidade ilimitada de processamento com velocidade instantânea, capacidade ilimitada de armazenamento, custo zero e não passível de falha), a fase de projeto envolve a modelagem de como o sistema será implementado com a adição dos requisitos tecnológicos ou não funcionais.



1.1 – A Fase de Projeto

O objetivo desta fase é incorporar a tecnologia aos requisitos essenciais do usuário, projetando o que será construído na implementação. Sendo assim, é necessário conhecer a tecnologia disponível e as facilidades do ambiente de software onde o sistema será desenvolvido e/ou implantado.

Independentemente do paradigma adotado, o projeto deve produzir:

- Projeto da Arquitetura do Software: definir os grandes componentes estruturais do software e seus relacionamentos.
- Projeto de Dados: projetar a estrutura dos dados necessária para implementar o software.
- Projeto de Interfaces: descrever como o software deverá se comunicar dentro dele mesmo (interfaces internas), com outros sistemas (interfaces externas) e com pessoas que o utilizam (interface com o usuário).
- Projeto Procedimental: refinar e detalhar a descrição procedimental dos componentes estruturais da arquitetura do software.

O projeto de software é um processo iterativo. Inicialmente, o projeto é representado em um nível alto de abstração. À medida que iterações ocorrem, os refinamentos conduzem a representações de menores níveis de abstração.

Uma especificação de projeto deve:

- Contemplar todos os requisitos explícitos contidos no modelo de análise e todos os requisitos implícitos desejados pelo cliente;
- Ser um guia legível e compreensível para aqueles que irão codificar, testar e manter o software.
- Prover um quadro completo do software, tratando aspectos funcionais, comportamentais e de dados, segundo uma perspectiva de implementação.

1.2 – Conceitos Básicos de Projeto

- **Níveis de Abstração:** o projeto deve considerar vários níveis de abstração, começando com em um nível mais alto, próximo da fase de análise. À medida que avançamos no processo de projeto, o nível de abstração deve ser reduzido.
- **Refinamento:** processo de elaboração, no qual o projeto vai sendo conduzido de níveis mais altos para níveis mais baixos de abstração.
- **Modularidade:** um projeto deve estruturar um sistema como módulos/componentes coesos e fracamente acoplados.
- **Ocultação de informações:** módulos / componentes são caracterizados pelas decisões de projeto que cada um deles esconde dos demais. Módulos devem ser projetados e especificados de modo que as informações neles contidas (dados e alguns procedimentos) sejam inacessíveis a outros módulos. ➔ Interface Contratual.

- **Independência Funcional:** módulos devem cumprir uma função bem estabelecida, minimizando interações com outros módulos. Esta característica é resultado direto da soma das demais características e pode ser medida usando dois critérios de qualidade: coesão e acoplamento.

1.3 – Princípios de Projeto

Em geral, um modelo de projeto (por exemplo, uma planta arquitetônica de uma casa) deve:

- começar representando a totalidade da coisa a ser construída;
- refinar para prover orientação para a construção de cada detalhe;
- prover diferentes visões da coisa.

Mais especificamente, o projeto de software deve:

- considerar abordagens alternativas com base nos requisitos do problema, recursos disponíveis e conceitos de projeto;
- ser rastreável ao modelo de análise;
- não “reinventar a roda”, isto é, reutilizar componentes;
- minimizar a distância conceitual e semântica entre o software e o mundo real;
- exibir uniformidade (estilo) e integração (interfaces entre componentes);
- ser estruturado para acomodar mudanças (alterabilidade);
- acomodar circunstâncias não usuais e, se necessário abortar o processamento, fazê-lo de modo elegante;
- apresentar nível de abstração superior ao código fonte. Projeto não é codificação;
- Ser passível de avaliação da qualidade;
- Ser revisado para minimizar erros semânticos.

1.4 – Qualidade do Projeto

Conforme citado anteriormente, a fase de projeto é responsável por incorporar requisitos tecnológicos aos requisitos essenciais. Assim, o projetista deverá estar atento aos critérios de qualidade que o sistema terá que atender. O modelo de qualidade definido na norma ISO/IEC 9126-1, utilizado como referência para a avaliação de produtos de software, define seis características de qualidade, desdobradas em sub-características (Rocha et al., 2001):

- **Funcionalidade:** refere-se à existência de um conjunto de funções que satisfaz às necessidades explícitas e implícitas e suas propriedades específicas. Tem como sub-características: adequação, acurácia, interoperabilidade, segurança de acesso e conformidade.
- **Confiabilidade:** diz respeito à capacidade do software manter seu nível de desempenho, sob condições estabelecidas, por um período de tempo. Tem como sub-características: maturidade, tolerância a falhas, recuperabilidade e conformidade.
- **Usabilidade:** refere-se ao esforço necessário para se utilizar um produto de software, bem como o julgamento individual de tal uso por um conjunto de usuários. Tem como sub-características: inteligibilidade, apreensibilidade, operacionalidade, atratividade e conformidade.
- **Eficiência:** diz respeito ao relacionamento entre o nível de desempenho do software e a quantidade de recursos utilizados sob condições estabelecidas. Tem como sub-características: comportamento em relação ao tempo, comportamento em relação aos recursos e conformidade.
- **Manutenibilidade:** concerne ao esforço necessário para se fazer modificações no software. Tem como sub-características: analisabilidade, modificabilidade, estabilidade, testabilidade e conformidade.
- **Portabilidade:** refere-se à capacidade do software ser transferido de um ambiente para outro. Tem como sub-características: adaptabilidade, capacidade para ser instalado, coexistência, capacidade para substituir e conformidade.

Xavier et al. (1995) utilizam outra classificação, indicando os seguintes aspectos a serem considerados:

- **Completeza:** diz respeito ao atendimento dos requisitos do cliente.
- **Desempenho:** refere-se ao uso otimizado dos recursos computacionais disponíveis (hardware, software e peopleware). Fatores que afetam o desempenho incluem:
 - ✓ Projeto físico de arquivos: deve minimizar o tempo de acesso a disco;
 - ✓ Reorganização de arquivos
 - ✓ Reorganização de índices
 - ✓ Limpeza de arquivos
 - ✓ Utilização da computação cliente-servidor: front-end local no servidor (interface com o usuário) e back-end no servidor (processamento e acesso a dados).

- **Segurança contra acessos indevidos:** é importante projetar:
 - ✓ Procedimentos de segurança, tais como controle de acesso (matriz classe de usuário x operações), criptografia e visões em bancos de dados;
 - ✓ Mecanismos de detecção de violações, tal como monitoração e arquivos de log.
- **Facilidade de uso:** diz respeito ao projeto de interface com o usuário;
- **Confiabilidade:** refere-se à preservação da disponibilidade do sistema e da integridade das informações armazenadas. No tocante a este critério de qualidade, deve-se ficar atento a:
 - ✓ Restrições de integridade: informações a serem armazenadas devem ser filtradas a partir das regras estabelecidas. Este filtro pode se dar via programa ou Sistema Gerenciador de Banco de Dados (SGBD).
 - ✓ Controle de Concorrência: bloqueio.
 - ✓ Recuperação de Falhas: prever possíveis falhas e definir sua forma de recuperação. Por exemplo, restaurar um banco de dados a um estado reconhecidamente correto após a ocorrência de uma falha.
 - Tipos de falhas:
 - Humana: digitação ou operação do sistema;
 - Transação: overflow, erro em tempo de execução, divisão por zero, ...
 - de Sistema: problemas no hardware, falta de energia, ...
 - de Comunicação: queda de linha, ...
 - de Software: erros de desenvolvimento
 - Prevenção de falhas:
 - em Arquivos: registros de cabeçalho (header) e de final (trailer);
 - Cópias de Segurança (backup);
 - Arquivos de Log para desfazer ou refazer transações;
 - Espelhamento (gravação simultânea em dois discos).
 - Detecção de falhas:
 - Auditoria: varredura de arquivos, apontando possíveis inconsistências.
 - Recuperação de falhas:
 - Desfazer ou refazer operações realizadas.

- **Manutenibilidade:** diz respeito à facilidade de alteração. Alterações podem ter origem em:
 - ✓ erros de especificação e/ou projeto;
 - ✓ novas necessidades do usuário;
 - ✓ alterações no ambiente tecnológico do usuário.Ações para aumentar a manutenibilidade:
 - ✓ Definir normas e padrões para interfaces com o usuário, relatórios, mensagens, codificação e nomenclatura de arquivos, módulos e programas.
 - ✓ Documentar
 - ✓ Modularizar
- **Economia:** o custo deve ser adequado ao escopo do software. Fatores que podem aumentar a economia incluem:
 - ✓ Reutilização: parcial de programa (cópia e alteração), de módulos, de módulos de biblioteca, de projeto.
 - ✓ Ferramentas CASE
 - ✓ Documentação

1.5 – Tecnologia Imperfeita e Requisitos Tecnológicos

Em função das limitações da tecnologia, devemos tomar decisões que adicionam os requisitos tecnológicos à essência do sistema.

Limitações da tecnologia:

- Custo
- Capacidade de armazenamento
- Velocidade de processamento
- Aptidão dos processadores: processador matemático, para comunicação entre processadores, etc.
- A tecnologia é passível de falha.

Impactos da tecnologia imperfeita:

- Uso de processadores diferentes / Distribuição / Comunicação
- Redundância: repetição de atividades e dados; inclusão de dados derivados (por exemplo, totalizadores)
- Novas atividades e funções acrescidas em função da imperfeição da tecnologia.

Levantamento dos Requisitos Tecnológicos junto aos Usuários:

- Localização geográfica de usuários / Transporte de dados
- Problemas operacionais nas atividades dos usuários
- Definição do ambiente de hardware e software de produção:
 - Frequência de disparo das atividades
 - Volume de dados (inicial, estimativa de crescimento, política de esvaziamento)
 - Tempo de resposta
 - Restrições técnicas (novo ambiente?)
 - Restrições ambientais (temperatura, etc.)
 - Requisitos de confiabilidade (tempo mínimo entre falhas)
 - Restrições de segurança (classes de usuários e acesso)
 - Interface com o usuário

Referências

- (Pressman, 2002) *Engenharia de Software*. Roger S. Pressman, tradução da 5ª edição, Mc Graw Hill, 2002.
- (Rocha et al., 2001) *Qualidade de Software: Teoria e Prática*. Ana Regina C. da Rocha, José Carlos Maldonado, Kival Chaves Weber, São Paulo: Prentice Hall, 2001.
- (Xavier et al., 1995) *Projetando com Qualidade a Tecnologia de Sistemas de Informação*. Carlos Magno da S. Xavier, Carla Portilho, Livros Técnicos e Científicos Editora, 1995.

2. Projeto Arquitetural

Referências: Cap.8 (Ruble, 1997), Caps. 2 e 3 (Xavier et al., 1995)

O modelo de arquitetura mapeia os requisitos essenciais da fase de análise em uma arquitetura técnica. Uma vez que muitas arquiteturas diferentes são possíveis, o propósito do projeto arquitetural é escolher a melhor configuração (ou talvez a “menos pior”). O processo de projetar a arquitetura inclui:

- a coleta de estatísticas de volumes de dados, frequência de disparo de eventos e expectativas de tempos de resposta para o modelo essencial,
- a documentação da topologia geográfica do negócio,
- a determinação da distribuição geográfica dos locais de computação,
- a definição do particionamento de dados e processos entre os locais de computação,
- a determinação do tráfego em rede e tamanho das bases de dados para várias configurações, e
- a validação do modelo de arquitetura contra o modelo essencial.

2.1 – Objetivos do Projeto Arquitetural

Até o projeto arquitetural, a questão do hardware ainda não foi tratada, já que na fase de análise pressupõe-se tecnologia perfeita. Este é o momento para resolver como este modelo ideal irá executar em uma plataforma restrita. É importante realçar que não existe a solução perfeita. O projeto da arquitetura é uma tarefa de negociação, onde se faz compromissos entre soluções sub-ótimas.

Em suma, o projeto arquitetural consiste na alocação do modelo essencial de requisitos em uma tecnologia específica, determinando que processos rodarão em quais processadores, onde os dados serão armazenados e quanta comunicação entre processadores será necessária. Ao término do projeto arquitetural, a equipe de desenvolvimento deve ter determinado:

- a distribuição geográfica dos requisitos computacionais,
- os componentes de hardware para as máquinas clientes,
- os componentes de hardware para as máquinas servidoras,
- a configuração e o número de camadas de hardware cliente-servidor,
- a plataforma de software de implementação, incluindo linguagens de codificação e apresentação (interface com o usuário), sistemas operacionais, os mecanismos e linguagens de comunicação em rede e sistema gerenciador de banco de dados,

- a localização ou localizações dos processos,
- a localização ou localizações dos dados físicos e
- as estratégias de sincronização para dados distribuídos geograficamente.

Muitas dessas decisões já terão sido tomadas previamente, seja por uma ordem da gerência, seja pelo fato da organização já possuir uma plataforma de hardware e software para implementação. Assim, o projeto de arquitetura é a busca por um meio menos objetável de atingir os requisitos do negócio, reconhecendo as limitações da plataforma a ser utilizada.

2.2 – A Arquitetura Cliente-Servidor

A computação cliente-servidor é um processamento cooperativo de informações de negócio por um conjunto de processadores, no qual múltiplos clientes geograficamente distribuídos iniciam requisições que são realizadas por um ou mais servidores centrais.

Primordialmente, o termo cliente-servidor é usado para descrever software que executa em mais de um hardware de modo a realizar uma tarefa do negócio. A separação de hardware é a norma em aplicações cliente-servidor, embora algumas pessoas utilizem o termo para descrever diferentes componentes de software se comunicando uns com os outros, ainda que rodando em uma mesma máquina. A distância entre processadores remotos varia desde computadores localizados na mesma sala ou prédio, até aqueles localizados em diferentes prédios, cidades ou mesmo espalhados pelo planeta.

Atualmente, há muitos tipos diferentes de processadores, que podem ser melhor utilizados para serem clientes ou servidores. Máquinas de grande porte (*mainframes*), por exemplo, são poderosas, mas requerem grande habilidade para operá-las e são sub-utilizadas para uma ampla variedade de requisições comuns. Computadores pessoais (PCs), por sua vez, são bem melhores para gerenciar de apresentações (interface com o usuário), necessitam de menos habilidades do usuário para operá-los e provêem processamento eficiente e barato para muitas das tarefas de uma organização.

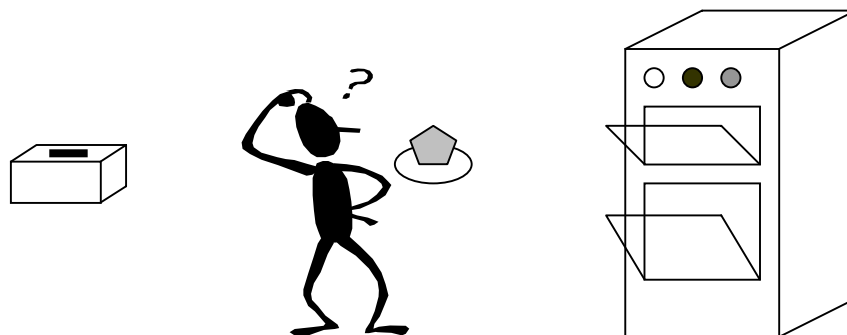


Figura 2.1 – Onde fazer uma torrada? Em uma torradeira ou em um fogão industrial?

2.2.1 - Camadas de Hardware Cliente-Servidor

O uso mais comum para arquiteturas cliente-servidor é explorar o poder dos PCs para gerenciar interfaces gráficas com o usuário, mantendo a integridade dos dados do negócio em uma máquina hospedeira central. Em sua forma mais simples, a arquitetura cliente-servidor envolve múltiplos clientes fazendo requisições para um único servidor, como mostra a figura 2.2. Este modelo mostra uma arquitetura de hardware em duas camadas (*two-tier*).

É fácil imaginar que, nesta arquitetura de hardware, a porção software de interface ficará no cliente, enquanto o armazenamento de dados deverá ser feito em um ou mais servidores centrais. Mas e o restante da aplicação? Não há respostas fáceis. Antes de explorar as possibilidades, vamos complicar um pouco mais a questão, introduzindo mais camadas na arquitetura cliente-servidor.

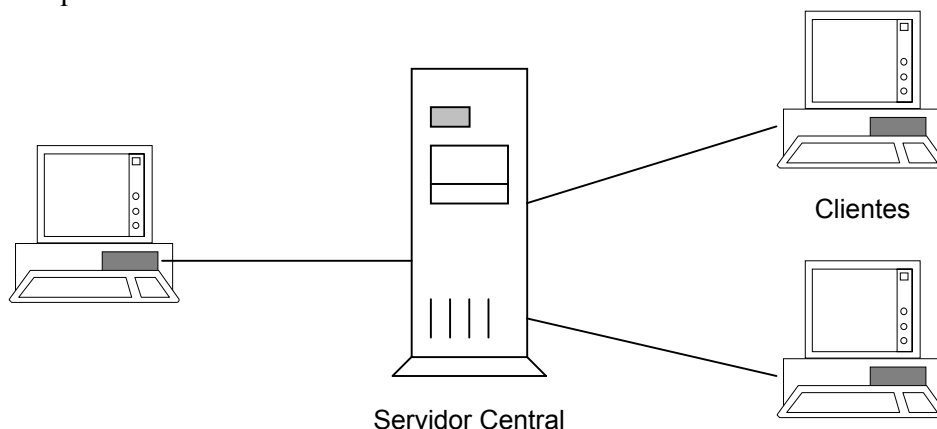


Figura 2.2 – Uma arquitetura de hardware cliente-servidor em duas camadas.

A figura 2.3 mostra uma arquitetura cliente-servidor em três camadas, na qual máquinas-cliente estão conectadas via uma rede local a um servidor local de aplicação que, por sua vez, se comunica com um servidor de dados central.

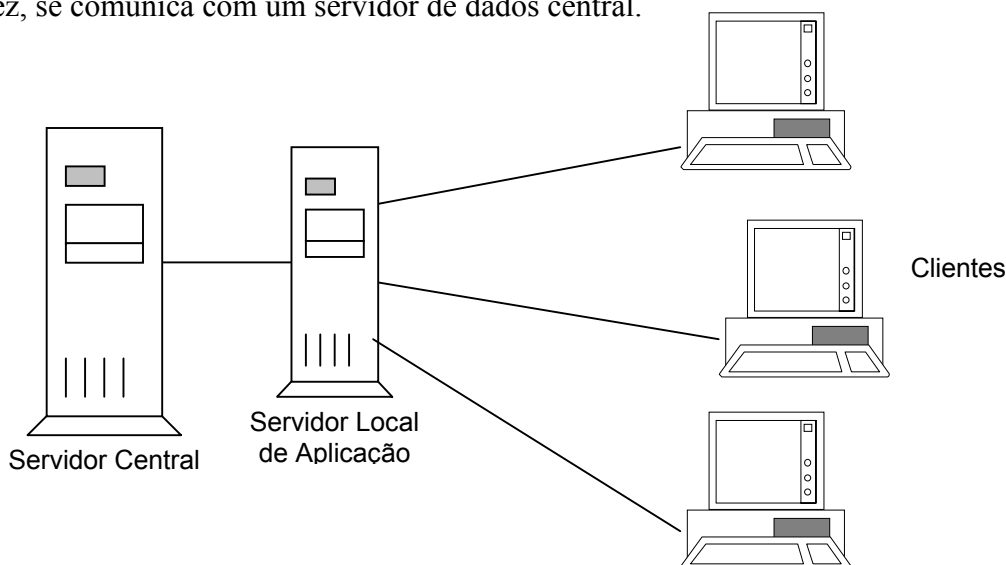


Figura 2.3 – Uma arquitetura de hardware cliente-servidor em três camadas.

Em uma arquitetura de três camadas, a noção de cliente e servidor começa a se tornar nebulosa. Um PC que hospeda uma aplicação de interface é certamente um cliente e a máquina hospedeira da base de dados é certamente um servidor. Mas e o servidor local de aplicação? Ele é algumas vezes um cliente e outras um servidor, dependendo da direção de comunicação. Esta arquitetura pode ser estendida para n camadas (n -tier), como mostra a figura 2.4. Nestes casos, a distinção entre cliente estrito e servidor estrito é destruída, tornando o termo um padrão conceitual.

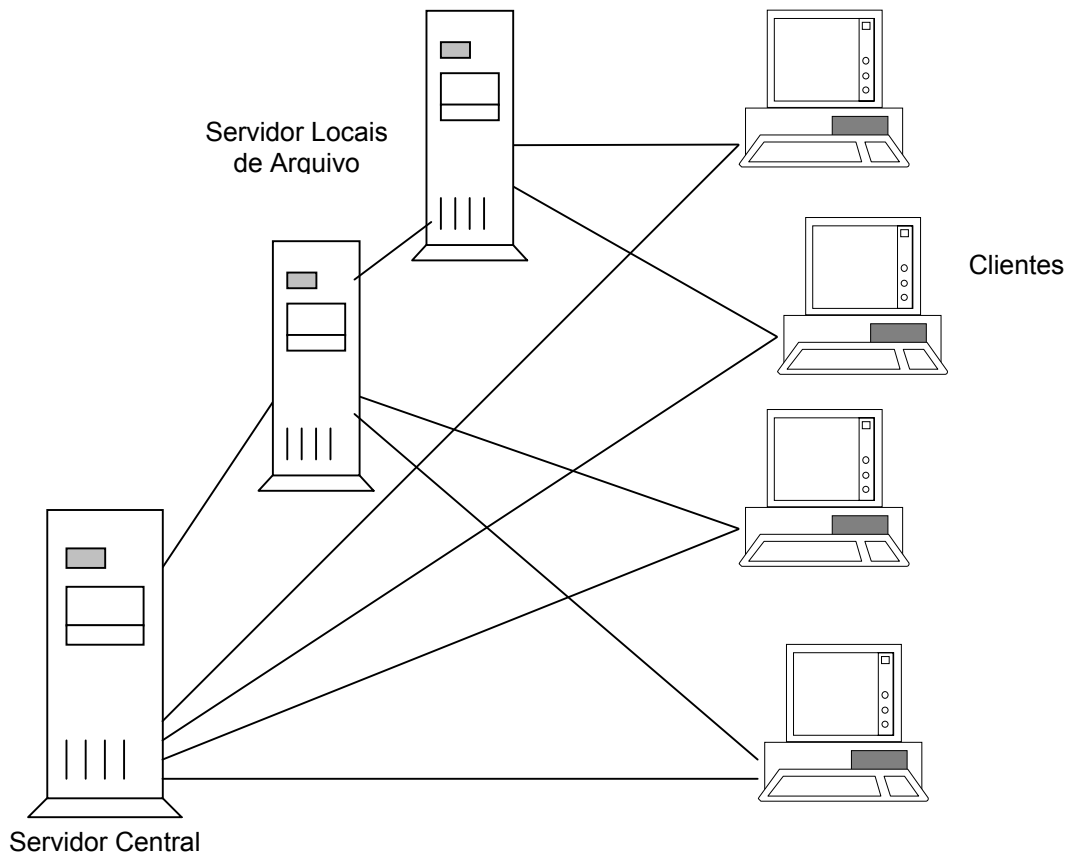


Figura 2.4 – Uma arquitetura de hardware cliente-servidor em n camadas.

2.2.2 - Camadas de Software Cliente-Servidor

Para discutir o desenvolvimento de software em uma arquitetura multi-camada de hardware, é necessário primeiro dissecar a aplicação de software em camadas. Os componentes de uma aplicação de negócio podem ser agrupados em pelo menos três categorias principais, como mostra a figura 2.5:

- **Camada de Apresentação:** é a camada mais externa do sistema de software. Sua função é capturar estímulos de eventos externos e realizar alguma edição dos dados de entrada. É encarregada também de apresentar respostas aos

eventos para o mundo exterior. Geralmente, é localizada em uma máquina cliente, tal como um PC, entretanto, esta não é uma regra rigorosa.

- **Camada de Lógica do Negócio:** contém o código que executa e impõe a política do negócio. Regras, regulamentos e cálculos são encontrados nesta camada. É a camada mais móvel, podendo ser localizada em clientes remotos, no servidor central ou em qualquer outro local intermediário.
- **Camada de Gerência de Dados:** provê acesso a dados corporativos. Gerencia requisições concorrentes de acesso às bases de dados, assim como a sincronização de elementos de dados distribuídos. Muito desta camada estará no mesmo local físico que os dados.

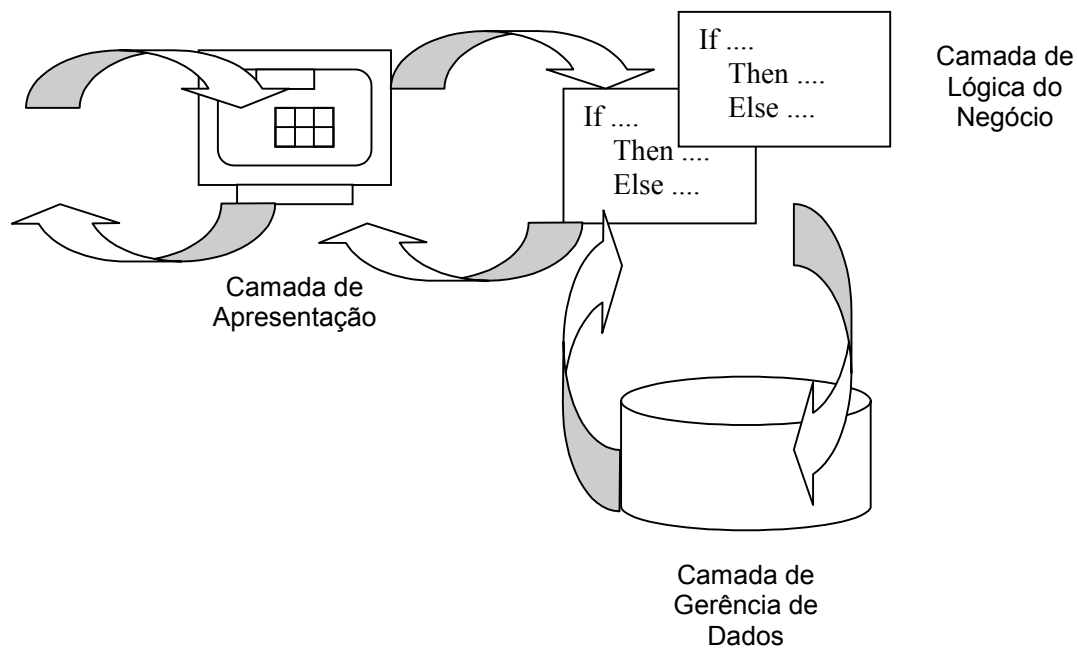


Figura 2.5 – Camadas de Software

2.2.3 - Cliente Pesado x Servidor Pesado

Uma classificação, relativamente incorreta, tem sido utilizada para designar a filosofia adotada em um projeto no que se refere à localização da maior parte da camada de lógica do negócio. O termo *cliente pesado* indica que a camada de lógica do negócio é colocada principalmente na máquina cliente e o servidor é dedicado ao acesso, distribuição e armazenamento de dados, respondendo a requisições de clientes. O termo *servidor pesado* descreve uma alocação de responsabilidades na qual o cliente é limitado à apresentação da interface e edição mínima, enquanto a maior parte da lógica de negócio e a imposição de regras são executadas no servidor central. É claro que esta é uma visão muito simplificada, já que arquiteturas cliente-servidor em *n*-camadas podem suportar uma gama de complexa de camadas de software.

A primeira geração de ferramentas populares de desenvolvimento de aplicações cliente-servidor com interfaces gráficas assumia uma arquitetura de hardware de duas camadas e encorajava uma abordagem de projeto arquitetural de software do tipo cliente pesado, onde a lógica do negócio era totalmente amarrada à camada de apresentação da aplicação. Estas mesmas ferramentas têm sido utilizadas, em um segundo momento, com uma filosofia do tipo servidor pesado, onde a lógica de negócio é fortemente mapeada no servidor de dados, na forma de *stored-procedures* e *triggers* em bancos de dados.

Respondendo às demandas por linguagens mais flexíveis de desenvolvimento, a segunda geração de ferramentas desta natureza reconhece a necessidade de separar a camada de lógica do negócio. Esta separação traz várias vantagens, como reusabilidade, portabilidade e manutenibilidade.

A reusabilidade tem sido apontada como a principal razão para a separação. Classes de apresentação são extremamente fáceis de reutilizar dada sua natureza altamente mecânica. Classes de negócio, por outro lado, são altamente complexas e podem desempenhar diferentes papéis dentro dos sistemas corporativos. A meta é criar classes que garantam todas as regras de negócio para uma particular classe de negócio e reutilizá-la em todos os contextos que lidem com a mesma. Esta abordagem é possível em uma filosofia de servidor pesado, mas impossível em uma de cliente pesado.

A portabilidade é a segunda razão para se efetuar a separação. A habilidade de mover a lógica de negócio ao longo da arquitetura cliente-servidor permite que se tire proveito de diferentes plataformas de hardware e software, sem ter que reescrever grandes porções do código.

Finalmente, a separação favorece a manutenção, já que concentra a lógica de negócio em uma camada única da arquitetura de software, facilitando a localização das alterações e evoluções a serem feitas no sistema.

2.3 – Coletando Estatísticas e Restrições

Determinar a arquitetura mais apropriada para um sistema envolve a quantificação da capacidade de computação necessária para o problema. O modelo de análise provê uma infra-estrutura confiável para a realização de estimativas dos requisitos de computação do sistema final. Assim, a primeira atividade do projeto arquitetural consiste em completar esta infra-estrutura, coletando informações estatísticas importantes.

2.3.1 – Coletando Estatísticas sobre o Modelo de Informação

A coleta de estatísticas sobre um modelo de informação – um modelo que descreve o sistema sob a ótica das informações que o mesmo precisa manipular, tal como um modelo de Entidades e Relacionamentos ou um Diagrama de Classes – é extremamente importante para a definição da arquitetura do software.

A meta principal deste passo é estimar o tamanho de uma base de dados. Para o projeto de um banco de dados relacional, duas informações são essenciais: o tamanho das

colunas e o número estimado de linhas que poderão acumular ao longo do tempo. Estes números são relativamente simples de serem obtidos.

Para determinar o tamanho das colunas, deve-se definir o tipo de dados para cada atributo da entidade/classe. Para estimar com certa precisão o número de bytes de uma única linha, basta somar o número de bytes para cada atributo da entidade/classe e adicionar o número de bytes da chave primária (se ela não for um atributo da entidade/classe) e das chaves estrangeiras, ou transpostas, que mapeiam os relacionamentos.

A estimativa do número de linhas esperadas para cada tabela não é tão simples. Algumas entidades/classes resultarão na criação de tabelas de controle, tais como Estado Civil e País. O número de instâncias nestas entidades/classes é relativamente fixo e, portanto, fácil de ser estimado. Contudo, é mais importante estimar o volume de dados para tabelas de transações, tais como Pedido e Itens de Pedido. Para estes casos, um modelo de eventos (por exemplo, modelo de casos de uso) pode nos dizer quais eventos criam instâncias destes tipos. Para estimar o tamanho dessas tabelas, é necessário saber quão freqüentemente o caso de uso ocorre. Esta estimativa deve ser obtida junto aos usuários. Além disso, é necessário determinar o período de retenção da informação sobre uma instância.

Em suma, colete as seguintes informações para cada entidade/classe do modelo de informação:

- tamanho estimado, em bytes, de uma instância, calculado pelo somatório dos tipos de dados de cada atributo, adicionado da estimativa para as chaves primária e estrangeiras,
- taxa de ocorrência dos casos de uso que criam novas instâncias da entidade/classe,
- período de retenção.

Estas estatísticas são usadas para estimar os recursos necessários para alojar adequadamente a base de dados. Mesmo se espaço em disco não for um problema para o projeto, elas são úteis para outros aspectos. Este problema torna-se mais complicado quando o repositório físico dos dados está geograficamente remoto do evento que necessita de seus dados.

Uma vez que é importante saber que tipo de operação um caso de uso pode realizar em uma entidade/classe, é útil produzir uma matriz CERA (Criar, Eliminar, Recuperar, Atualizar) Caso de Uso/Classe (ou Caso de Uso/Entidade), mostrando se, por ocasião da ocorrência do caso de uso, o sistema necessitará criar, atualizar, recuperar ou eliminar instâncias da entidade/classe.

Caso de Uso\Classe	Cliente	Pedido	Item de Pedido
Efetuar pedido	R	C	C
Cancelar pedido	R	RE	RE
Alterar pedido	R	RA	CERA

Figura 2.6 – Exemplo de uma Matriz CERA Caso de Uso/Classe.

2.3.2 – Coletando Estatísticas sobre o Modelo de Eventos

Um modelo de eventos é aquele que captura as funcionalidades de um sistema segundo uma perspectiva exterior, isto é, dos usuários. A lista de eventos da Análise Essencial ou o modelo de casos de uso na Orientação a Objetos são exemplos de modelos de eventos.

Alguns casos de uso são bem mais críticos do que outros no que diz respeito à importância do sistema ser hábil para responder rapidamente. Pode-se fazer anotações com estatísticas sobre o modelo de eventos para capturar a frequência de ocorrência (ou disparo) e o tempo de resposta desejado para um evento. O objetivo é identificar os eventos críticos, já que eles serão objeto de exame minucioso. Para esses, deve-se levantar a frequência média de ocorrência, a taxa de pico e o período de tempo do pico.

A frequência média de disparo nos diz apenas o nível normal de ocorrência para um evento. Esta informação é suficiente para eventos uniformes, isto é, aqueles cuja frequência de disparo tende a ser mesma ao longo do todo tempo. Para eventos irregulares, contudo, é necessário conhecer as taxas de pico e o período em que estes picos ocorrem.

A principal questão a ser colocada sobre este assunto é: o sistema tem que ser dimensionado para tratar picos, de modo a sempre atender satisfatoriamente às necessidades dos usuários? Se o sistema for dimensionado pela capacidade de pico, provavelmente ele ficará ocioso grande parte do tempo. Por outro lado, se for dimensionado pela média, nos momentos de pico não atenderá totalmente às necessidades dos usuários. Esta não é uma decisão fácil e não deve ser tomada apenas pelo pessoal de desenvolvimento. Ao contrário, esta decisão cabe, principalmente, ao cliente. De maneira geral, a menos que o custo seja proibitivo, a maioria das organizações prefere gastar mais dimensionando o sistema pelo pico, evitando inconvenientes para os usuários.

Uma alternativa criativa, mas nem sempre possível, consiste em tentar suavizar o pico de alguns eventos, oferecendo incentivos para os usuários utilizarem o sistema fora do período de pico.

Não é necessário analisar todos os eventos de um modelo de eventos, coletando estatísticas para todo o sistema. Devemos nos concentrar nos principais casos de uso do negócio, aqueles com os maiores impactos sobre o cliente, com os maiores volumes de dados e com as localizações mais remotas geograficamente.

Outro aspecto muito importante sobre casos de uso diz respeito aos requisitos de desempenho. Uma característica de qualidade recorrente para sistemas de informação é o tempo de resposta. O tempo de resposta máximo aceitável deve ser definido para cada evento/caso de uso. Novamente, uma boa dose de bom senso é necessária. A equipe do projeto deve concentrar seus esforços sobre os eventos de negócio mais importantes, ou seja, aqueles que ocorrem com maior frequência e têm mais impactos para os clientes e usuários.

O tempo de resposta pode ser medido tanto no nível de evento do negócio quanto no nível de diálogo. No nível de evento do negócio, a métrica de tempo de resposta mede o tempo entre a recepção do estímulo inicial do evento até a emissão da resposta final. No nível de diálogo, a métrica de tempo de resposta mede o tempo que o sistema leva para responder a um diálogo/consulta, que pode ser apenas uma porção do caso de uso total. Obviamente, quando o tempo de resposta é expresso no nível de diálogo, o projetista se depara com uma situação bem mais restrita.

2.3.3 – Determinando Requisitos de Distribuição Geográfica

O próximo passo do projeto arquitetural consiste em examinar a distribuição geográfica dos casos de uso, o que conduz naturalmente à distribuição necessária ao acesso dos dados. Juntos, a frequência de disparo dos casos de uso, volume de dados, restrições de tempo de resposta e a distribuição geográfica do negócio formarão a base para se determinar uma arquitetura aceitável para o sistema em questão.

Algumas matrizes podem ser usadas para mapear o modelo de análise na topologia de localizações do negócio. Uma matriz Caso de Uso/Localização do Negócio, como a mostrada na figura 2.7, juntamente com uma matriz Caso de Uso/Classe (ou Evento/Entidade), como a mostrada na figura 2.6, permitem mapear as necessidades de distribuição geográfica da computação de uma organização.

Caso de Uso /Localização	Internet	Vitória	Linhares	Cachoeiro
Efetuar pedido		X	X	X
Cancelar pedido	X	X		
Alterar pedido	X	X		

Figura 2.7 – Exemplo de uma Matriz Caso de Uso/Localização do Negócio.

A partir dessas duas matrizes, uma terceira pode ser derivada. Uma vez que se conhece a distribuição geográfica dos eventos e os requisitos de acesso a dados de cada evento, é possível derivar a distribuição geográfica dos requisitos de acesso a dados, a matriz CERA Localização do Negócio/Classe (Entidade), como mostra o exemplo da figura 2.8. Esta matriz mostra exatamente que localizações do negócio precisam criar, recuperar, atualizar ou eliminar instância de cada uma das entidades/classes do sistema. Com esta informação é possível começar a avaliar potenciais soluções arquiteturais.

Localização/Classe	Cliente	Pedido	Item de Pedido
Internet	R	ERA	CERA
Vitória	R	CERA	CERA
Linhares	R	C	C
Cachoeiro	R	C	C

Figura 2.8 – Exemplo de uma Matriz CERA Localização/Classe.

Uma variação útil da matriz CERA Caso de Uso/Classe (Evento/Entidade) é a matriz de Aceitação Caso de Uso/Classe (ou Evento/Entidade), mostrada na figura 2.9. Ela mostra quão atualizada uma base de dados deve estar para atender um determinado caso de uso. O valor zero indica que a base de dados deve refletir instantaneamente a atualização feita mais recentemente.

Caso de Uso/Classe	Cliente	Pedido	Item de Pedido
Efetuar pedido	48	0	0
Cancelar pedido	48	0	0
Alterar pedido	48	0	0

Figura 2.9 – Exemplo de uma Matriz de Aceitação Caso de Uso/Classe (em horas).

Da mesma forma que as matrizes Caso de Uso/Localização e CERA Caso de Uso/Classe foram combinadas para derivar uma matriz CERA Localização/Classe, as matrizes Caso de Uso/Localização e de Aceitação Caso de Uso/Classe podem ser combinadas para derivar uma matriz de Aceitação Localização/Classe para mostrar a necessidade de acesso a dados remotos, como mostra a figura 2.10.

Localização/Classe	Cliente	Pedido	Item de Pedido
Internet		0	0
Vitória		0	0
Linhares		0	0
Cachoeiro		0	0

Figura 2.10 – Exemplo de uma Matriz de Aceitação Localização/Classe.

Esta matriz mostra uma visão muito geral, já que não revela o fato de apenas certos atributos serem acessados em certas localizações. Esta visão pode ser refinada, construindo matrizes para cada entidade/classe, mostrando que atributos são usados em que localizações. A análise da distribuição de dados no nível de atributo é necessária apenas quando os problemas forem muito complexos e distribuídos.

2.4 – Estratégias de Acesso Oportuno a Dados

Uma vez levantadas as necessidades de acesso a dados de cada localização do negócio, pode-se avaliar qual a melhor estratégia para a distribuição dos mesmos. A seguir, as principais estratégias são discutidas.

2.4.1 – Uma Única Base de Dados Centralizada

A primeira opção de solução a ser considerada para a distribuição de dados consiste em, contraditoriamente, não distribuí-los de fato. Uma única cópia dos dados é mantida em uma localização central e todas as aplicações que necessitem acessá-los devem fazer suas consultas e atualizações no servidor central. Os benefícios são numerosos:

- É fácil fazer cópias de segurança dos dados quando uma única cópia existe.
- O projeto do sistema total é menos complicado, por exemplo, a segurança é mantida centralmente e nenhuma rotina de sincronização é requerida.
- Os dados são sempre atuais.
- Não há dados redundantes ao longo das localizações do negócio.

Entretanto, as desvantagens podem ser significativas no desenvolvimento de uma aplicação geograficamente remota:

- Até os dias atuais, muitas tecnologias de comunicação de dados não são ainda rápidas o suficiente ou são muito caras para aplicações de grande escala. As estatísticas coletadas de volume de dados e frequência de disparo de eventos quando comparadas com a capacidade de comunicação da rede podem dizer se o acesso remoto a dados é viável.
- Tem-se um grande problema se o servidor central ou as linhas de comunicação caírem. Neste caso, os locais remotos ficaram efetivamente sem processamento.

Desempenho inaceitável e risco de queda são os dois fatores que fazem com que negócios muitas vezes descartem bases de dados centralizadas.

2.4.2 – Dados Descentralizados e Replicados

De um lado diretamente oposto, a base de dados pode ser completamente replicada em todos os locais que dela necessitem. Atualizações em um local podem ser irradiadas (*broadcast*) para outros locais em tempo real. Com esta estratégia, há vários benefícios óbvios:

- O projeto das aplicações locais é simplificado por ter acesso a dados locais.
- O tempo de resposta para cada transação não é onerado pelo alto tráfego em rede.
- Incentiva proprietários locais de dados e provê acesso local rápido.

As desvantagens, contudo, envolvem muitas complicações:

- O tráfego global na rede aumenta devido à replicação de dados em todos os locais.
- Rotinas complexas de sincronização são necessárias para manter as várias cópias da base de dados atualizadas.
- Podem surgir problemas se o mesmo registro for atualizado em dois locais.
- Se um dos servidores cair ou o software de replicação falhar, pode ser difícil reconstruir o conjunto dos dados e reaplicar atualizações na ordem correta.
- Qual base de dados é a principal? Procedimentos de back-up tornam-se mais complexos.
- Dados completamente replicados podem conduzir a redundância desnecessária de dados.

Fragmentação

Um compromisso é freqüentemente sugerido entre a centralização e a replicação totais. A distribuição dos dados é otimizada de modo que apenas os dados necessários por cada local são mantidos locais. Esta estratégia é chamada fragmentação e pode ser vertical ou horizontal.

A fragmentação vertical ocorre quando apenas certas tabelas ou colunas são fisicamente distribuídas a locais remotos. Cada localização possui apenas aquelas tabelas ou colunas que são requeridas pelos eventos que ocorrem no mesmo. Isto reduz tráfego em rede, pois apenas os elementos de dados necessários precisam ser sincronizados com outros locais. Entretanto, esta estratégia pode ser bastante complexa para gerenciar. Os procedimentos de replicação devem ser capazes de sincronizar atualizações coluna-por-coluna em diferentes locais.

A fragmentação horizontal ocorre quando apenas certas linhas em uma tabela são fisicamente distribuídas a locais remotos. Esta estratégia é empregada tipicamente quando localizações têm seus próprios dados que não são manipulados em outras localizações. Cada localização tem sua cópia completa do esquema de banco de dados. As estruturas das tabelas em cada localização são idênticas, mas as linhas de dados que povoam estas tabelas podem ser diferentes. Geralmente, há uma base de dados principal que contém todos os registros. Assim como a fragmentação vertical, a horizontal diminui o tráfego na rede, eliminando transferências de dados desnecessárias. Contudo, o processo de sincronização é também de alguma forma complicado, principalmente quando diferentes locais compartilham alguns registros.

Uma combinação dos dois tipos de fragmentação apresentados anteriormente pode ser utilizada. Bases de dados distribuídas compartilham os mesmos tipos de entidades lógicas, mas possuem diferentes colunas e linhas. Cada local possui apenas as colunas e linhas que são realmente necessárias para os eventos que ocorrem ali. É fácil perceber que tal estratégia quando levada ao extremo pode ser muito difícil de gerenciar.

2.5 – Sumário da Determinação da Distribuição Geográfica

Os modelos criados durante a fase de análise provêm uma riqueza de informação para a determinação dos requisitos de distribuição geográfica de um sistema. Podemos começar coletando estatísticas para determinar o tamanho total da base de dados. Esta informação é crítica, não só para a determinação da necessidade de espaço em disco, mas também para situações onde os dados são fisicamente distribuídos. Neste caso, o tamanho dos registros e a frequência de disparo dos eventos permitirão estimar o tráfego em rede. Estatísticas capturadas do modelo de eventos incluem a frequência média de disparo dos eventos, taxas de pico para eventos irregulares e expectativas de tempo de resposta.

De posse destas estatísticas, o próximo passo consiste em aplicá-las à topologia de localização do negócio. Várias matrizes têm se mostrado úteis para modelar este aspecto do sistema, tais como matrizes Casos de Uso/Localização Geográfica, matrizes CERA Caso de Uso/Classe e matrizes CERA Localização do Negócio/Classe.

Uma vez que os requisitos de acesso a dados foram determinados, informação adicional pode ajudar a equipe de projeto a efetuar escolhas arquiteturais sólidas. Matrizes de aceitação, por exemplo, mostram quão “antiga” uma entidade pode ser para um dado evento ocorrendo em uma localização. Isto pode ajudar a decidir se uma rotina de sincronização em tempo real é necessária ou se uma rotina de atualização *batch* é suficiente.

Conhecidos o tamanho dos registros, a taxa dos eventos que criam, lêem, atualizam e eliminam estes registros e a distância entre os locais do negócio, a equipe pode, enfim, explorar suas opções arquiteturais. Estratégias de distribuição de dados variam desde completamente centralizada a completamente distribuída. Qualquer que seja a escolha feita, ela deve respeitar, acima de tudo, as necessidades dos usuários.

2.6 – Segurança

A segurança contra acessos indevidos tem como objetivo não permitir que pessoas não autorizadas tenham acesso a eventos ou aos dados.

Para controlar o acesso, é necessário identificar, autenticar e autorizar usuários. A identificação se dá através de um código unívoco capaz de identificar apenas um usuário. A autenticação consiste em comprovar que o usuário é mesmo quem ele diz ser na identificação, sendo feita, normalmente, por uma senha. Finalmente, a autorização é dada ao usuário, ou a uma classe de usuários, dando acesso a determinados eventos, dados para leitura/escrita e tipos de transações.

Um caso de uso de cunho tecnológico tem de ser adicionado à lista de eventos/modelo de casos de uso, para permitir definir classes de usuários e seus perfis, incluir, excluir ou alterar usuários, além, é claro, das atividades de identificação, autenticação e autorização de usuários. Uma matriz Caso de Uso / Classe de Usuário pode ser utilizada para documentar o nível de autorização adotado no projeto, como mostra a figura 2.11.

Caso de Uso/Classe Usuário	Cliente	Gerente	Funcionário
Efetuar pedido		X	X
Cancelar pedido	X	X	X
Alterar pedido	X	X	X
Solicitar relatório de pedidos		X	

Figura 2.11 – Exemplo de uma Matriz de Caso de Uso/Classe de Usuário.

Outra atividade que deve ser adicionada ao projeto é a detecção da ocorrência de violações. Muitas vezes só percebemos que um sistema foi violado após o ocorrido. Quando for necessário maior nível de segurança, é importante criar um procedimento de monitoração que registre os usuários que acessaram o sistema e os casos de uso por eles realizados. O arquivo que guarda essas informações é chamado de histórico (*log*). Para consulta a esse arquivo, é necessário criar mais um caso de uso de cunho tecnológico, a auditoria.

2.7 – Modelo de Tarefas

Mesmo quando um sistema não será distribuído geograficamente, é útil definir tarefas ou unidades de execução, isto é, conjunto de casos de uso essenciais e tecnológicos agrupados em uma unidade, do ponto de vista de execução pelo sistema operacional.

O padrão para empacotamento em tarefas é o agrupamento de casos de uso por tipo de processador, isto é, todos os casos de uso que devam rodar em um mesmo tipo de máquina são agrupados em uma tarefa. Porém, em alguns casos, podem ser necessárias mais tarefas, tal como:

- Quando existir uma classe de usuários específica, cujos casos de uso autorizados sejam de seu único acesso e interesse.
- Quando usuários estão dispersos geograficamente, como discutido anteriormente.
- Não há memória disponível suficiente para que a tarefa rode eficientemente.
- Casos de uso são realizados uma única vez (ou muito poucas vezes), tais como conversão de dados e instalação do sistema.

A principal estratégia para agrupar casos de uso em tarefas consiste em fazer um levantamento sobre o modo de utilização do sistema, o tempo de resposta esperado, o tamanho e número de casos de uso, aspectos de segurança e facilidade de uso, como discutido para o caso da distribuição geográfica.

2.8 – Atividades Tecnológicas

Em decorrência das imperfeições e limitações da tecnologia (requisitos tecnológicos), novos casos de uso podem ser adicionados ao modelo de eventos, tais como, as atividades de segurança anteriormente relacionadas, limpeza, reorganização, cópia e restauração de arquivos e facilidades de ajuda (*help*).

É bom lembrar que todo caso de uso tecnológico projetado é decorrente de um requisito tecnológico acrescido ao sistema, mas nem todo requisito tecnológico necessariamente implica na criação de um caso de uso tecnológico. Algumas destas funcionalidades podem ser melhor caracterizadas como procedimentos tecnológicos, constando no interior da especificação de um ou mais casos de uso (essenciais ou tecnológicos) do sistema.

Referências:

- (Ruble, 1997) *Practical Analysis and Design for Client/Server and GUI Systems*. David A. Ruble, Yourdon Press Computing Series, 1997.
- (Xavier et al., 1995) *Projetando com Qualidade a Tecnologia de Sistemas de Informação*. Carlos Magno da S. Xavier, Carla Portilho, Livros Técnicos e Científicos Editora, 1995.

3 - Projeto de Interface com o Usuário

A maioria dos sistemas atuais é desenvolvida para ser utilizada por pessoas. Assim, um aspecto fundamental no projeto de sistemas é a interface com o usuário (IU). Nesta etapa do projeto, são definidos os formatos de janelas e relatórios, entre outros, sendo a prototipagem bastante utilizada, buscando auxiliar o desenvolvimento e a seleção dos mecanismos reais de interação. A IU capta como um usuário comandará o sistema e como o sistema apresentará as informações a ele.

O princípio básico para o projeto de interfaces com o usuário, em geral, é o seguinte: “Conheça o usuário e as tarefas”. O projeto de interface com o usuário envolve não apenas aspectos de tecnologia (facilidades para interfaces gráficas, multimídia, etc), mas principalmente o estudo das pessoas. Quem é o usuário? Como ele aprende a interagir com um novo sistema? Como ele interpreta uma informação produzida pelo sistema? O que ele espera do sistema? Estas são apenas algumas das muitas questões que devem ser levantadas durante o projeto da interface com o usuário (Pressman, 2002). De maneira geral, o projeto de interfaces com o usuário segue o seguinte processo global, como mostra a figura 3.1:

1. *Delinear as tarefas necessárias para obter a funcionalidade do sistema*: este passo visa capturar as tarefas que as pessoas fazem normalmente no contexto do sistema e mapeá-las em um conjunto similar (mas não necessariamente idêntico) de tarefas a serem implementadas no contexto da interface homem-máquina.
2. *Estabelecer o perfil dos usuários*: A interface do sistema deve ser adequada ao nível de habilidade dos seus futuros usuários. Assim, é necessário estabelecer o perfil destes potenciais usuários e classificá-los segundo aspectos como nível de habilidade, nível na organização e membros em diferentes grupos. Uma classificação possível considera os seguintes grupos:
 - *Usuário Novato*: não conhece os mecanismos de interação requeridos para utilizar a interface eficientemente (conhecimento sintático) e conhece pouco a aplicação em si, isto é, entende pouco as funções e objetivos do sistema (semântica da aplicação);
 - *Instruído, mas intermitente*: possui um conhecimento razoável da semântica da aplicação, mas tem relativamente pouca lembrança das informações sintáticas necessárias para utilizar a interface;
 - *Instruído e freqüente*: possui bom conhecimento tanto sintático quanto semântico e buscam atalhos e modos abreviados de interação.
3. *Considerar aspectos gerais de projeto de interface*, tais como tempo de resposta, facilidades de ajuda, mensagens de erro, tipos de comandos, entre outros.
4. *Construir protótipos* e, em última instância, implementar as interfaces do sistema, usando ferramentas apropriadas. A prototipação abre espaço para uma abordagem iterativa de projeto de interface com o usuário, como mostra abaixo. Entretanto, para tal é imprescindível o suporte de ferramentas para a construção

de interfaces, provendo facilidades para manipulação de janelas, menus, botões, comandos, etc...

5. *Avaliar o resultado*: Coletar dados qualitativos e quantitativos (questionários distribuídos aos usuários do protótipo).

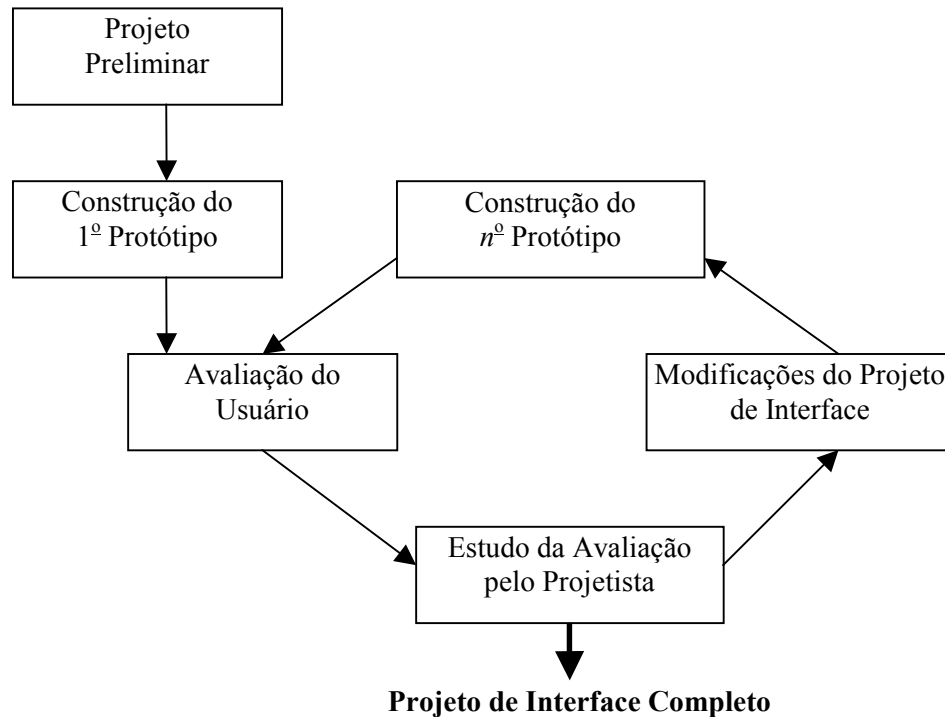


Figura 3.1 - Abordagem Iterativa para o Projeto de Interface com o Usuário.

3.1 – Aspectos Gerais de Interface com o Usuário

Com relação aos aspectos gerais de um projeto de interface, citados no item 3, devemos considerar, pelo menos, o seguinte conjunto:

- **Tempo de Resposta**: É importante mostrar o progresso do processamento para os usuários, principalmente para eventos com tempo de resposta longo ou com grande variação de tempos de resposta.
- **Facilidade de Ajuda (Help)**: Definir:
 - ✓ quando estará disponível e para que funções do sistema;
 - ✓ como ativar (botão, tecla de função, menu);
 - ✓ como representar (janela separada, local fixo da tela);
 - ✓ como retornar à interação normal (botão, tecla de função);
 - ✓ como estruturar a informação (estrutura plana, hierárquica, hipertexto).

- Mensagens de Erro e Avisos: Devem:
 - ✓ descrever o problema com um vocabulário passível de entendimento pelo usuário;
 - ✓ prover assistência para recuperar o erro;
 - ✓ indicar quaisquer conseqüências negativas do erro;
 - ✓ ser acompanhadas de uma “dica” visual ou sonora;
 - ✓ ser sem censura ao usuário.
- Tipos de Comandos: Em muitas situações deve-se prover ao usuário a opção de utilização de comandos. Nestes casos, definir e avaliar:
 - ✓ se toda opção de menu terá um comando correspondente;
 - ✓ a forma do comando (controle de seqüência (^Q), teclas de função (F1), palavra);
 - ✓ quão difícil é aprender e lembrar o comando;
 - ✓ possibilidade de customização de comandos (macros);
 - ✓ padrões para todo sistema e conformidade com outros padrões.

Algumas orientações adicionais devem ser consideradas e são listadas na seqüência.

Diretrizes Gerais:

- Seja consistente (formato consistente para seleção de menus, entrada de comandos, apresentação de dados, ...);
- Ofereça retorno significativo ao usuário (comunicação com o usuário);
- Peça confirmação para ações destrutivas (deletar arquivo, sobrepor informação, terminar a aplicação,...);
- Permita reversão fácil da maioria das ações (função UNDO);
- Reduza a quantidade de informação que precisa ser memorizada entre ações;
- Busque eficiência no diálogo (movimentação, teclas a serem apertadas);
- Trate possíveis erros do usuário (o sistema deve se proteger de erros, casuais ou não, provocados pelo usuário);
- Classifique atividades por função e organize geograficamente a tela de acordo (menus pull-down);
- Proveja facilidades de ajuda sensíveis ao contexto;
- Use verbos de ação simples ou frases verbais curtas para nomear funções e comandos.

Diretrizes para Apresentação de Informação

- Mostre apenas informações relevantes ao contexto corrente;
- Use formatos de apresentação que permitam assimilação rápida da informação (gráficos, figuras, etc.);
- Use rótulos consistentes, abreviaturas padrão e cores previsíveis;
- Produza mensagens de erro significativas;
- Projete adequadamente o layout de informações textuais (letras maiúsculas e minúsculas, indentação, agrupamento de texto, etc.);
- Use janelas para separar diferentes tipos de informação;
- Use formas de representação análogas às do mundo real para facilitar a assimilação da informação (figuras, cores, etc.).

Diretrizes para a Entrada de Dados:

- Minimize o número de ações de entrada requeridas (seleção de dados a partir de um conjunto pré-definido de valores de entrada, macros, etc.);
- Mantenha consistência entre apresentação e entrada de dados (características visuais: tamanho do texto, cor, localização, etc.);
- Permita ao usuário customizar a entrada (comandos customizados, dispensar algumas mensagens de aviso e verificações de ações, etc.);
- Flexibilize a interação, permitindo afiná-la ao modo de entrada preferido do usuário (comandos, botões, *plug-and-play*, digitação, etc.);
- Desative comandos inapropriados para o contexto das ações correntes;
- Proveja ajuda para assistir todas as ações de entrada de dados;
- Proveja valores *default*, sempre que possível;
- Nunca requiera que o usuário entre com uma informação que possa ser adquirida automaticamente pelo sistema ou computada por ele.

Referências

(Pressman, 2002) *Engenharia de Software*. Roger S. Pressman, tradução da 5ª edição, Mc Graw Hill, 2002.

4. Projeto de Bancos de Dados Relacionais

Um aspecto fundamental da fase de projeto consiste em estabelecer de que forma serão armazenados os dados do sistema. Em função da plataforma de implementação, diferentes soluções de projeto devem ser adotadas. Isto é, se o software tiver de ser implementado em um banco de dados hierárquico, por exemplo, um modelo hierárquico deve ser produzido, adequando a modelagem conceitual de dados (ER ou diagrama de classes) a esta plataforma de implementação.

Atualmente, a plataforma de implementação para armazenamento de dados mais difundida é a dos Bancos de Dados Relacionais e, portanto, neste texto, discutiremos apenas o modelo relacional.

4.1 - O Modelo Relacional

Em um modelo de dados relacional, os conjuntos de dados são representados por tabelas de valores. Cada tabela, denominada de relação, é bidimensional, sendo organizada em linhas e colunas. Este modelo está fortemente baseado na teoria matemática sobre relações, daí o nome relacional.

Ex:

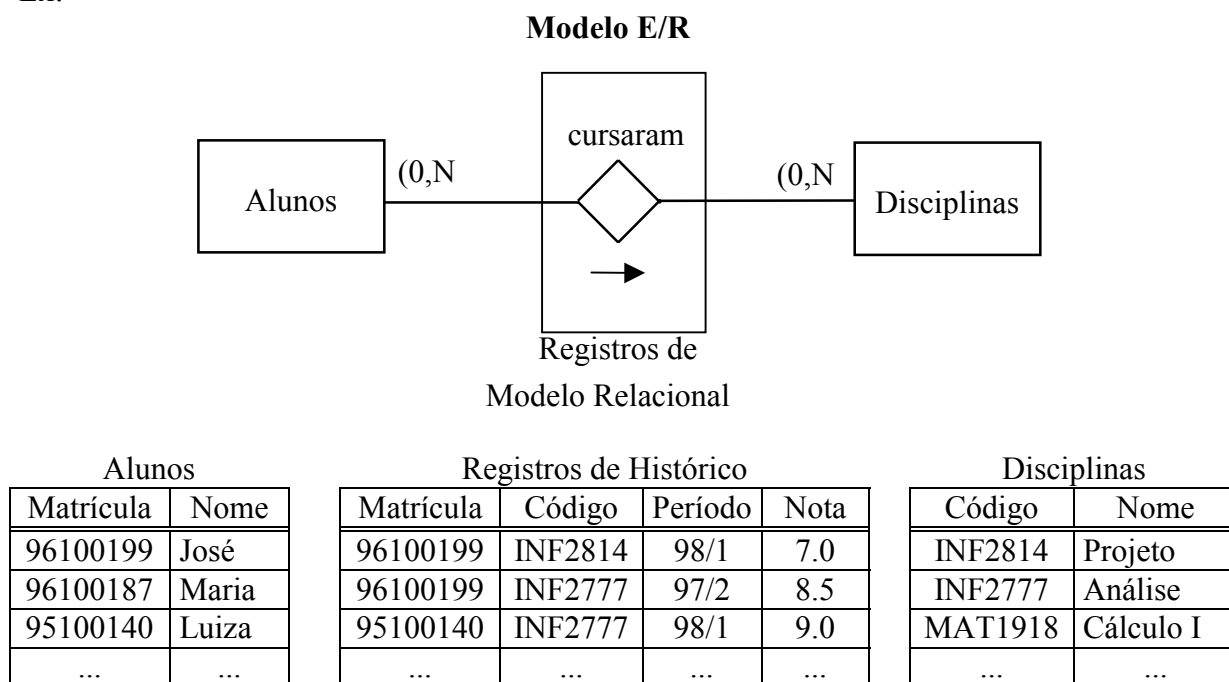


Figura 4.1 - Exemplo de um Modelo Relacional derivado a partir de um Modelo E/R.

Conceitos Básicos

- **Relação:** Tabela de valores bidimensional organizada em linhas e colunas. Representa um conjunto de entidades do Modelo E/R ou uma classe em um Diagrama de Classes.
Ex: Relação Funcionários.

Matrícula	Nome	CPF	Endereço	Dt-Nasc	Dt-Adm
0111	Marcos	17345687691	Vila Velha	11/04/66	20/08/86
0208	Rita	56935101129	Vila Velha	21/02/64	18/03/90
0789	Mônica	81176628911	Vitória	01/11/70	15/07/92
1589	Márcia	91125769120	Serra	20/10/80	01/02/98
...

- **Grau da Relação:** Número de colunas da tabela. Ex: 6
- **Linha (Tupla):** Representa uma entidade do conjunto de entidades, ou um objeto de uma classe.
Ex: Funcionário 0789 do conjunto de Funcionários.
- **Colunas:** Representam os vários atributos do conjunto de entidades ou classe.
Ex: Matrícula, Nome, CPF, Endereço, Dt-Nasc, Dt-Adm.
- **Célula:** Item de dado elementar da linha i, coluna j.
Ex: Vitória (linha 3, coluna 4)
01/02/98 (linha 4, coluna 6)
- **Chave Primária:** Atributo ou combinação de atributos que possuem a propriedade de identificar de forma única uma linha da tabela. Corresponde a um atributo determinante do Modelo Conceitual.
Ex: Matrícula
- **Chaves Candidatas:** Ocorrem quando em uma relação existe mais de uma combinação de atributos possuindo a propriedade de identificação única.
Ex: Matrícula, CPF
- **Chave Estrangeira:** Ocorre quando um atributo de uma relação for chave primária em outra relação.
- **Ligações:** Representam os relacionamentos do Modelo E/R ou as associações em um Diagrama de Classes. A ligação entre duas relações é feita, em geral, transportando-se a chave de uma relação para outra (item transposto), como ilustra a figura 4.2. Neste exemplo, a chave da tabela Departamentos foi transposta para a tabela Funcionários. Assim, “Departamentos” é denominada relação origem e “Funcionários” relação destino. No caso de relacionamentos muitos-para-muitos, é necessário criar uma nova tabela, dita **Tabela Associativa**, que deverá ter as chaves das duas tabelas relacionadas.

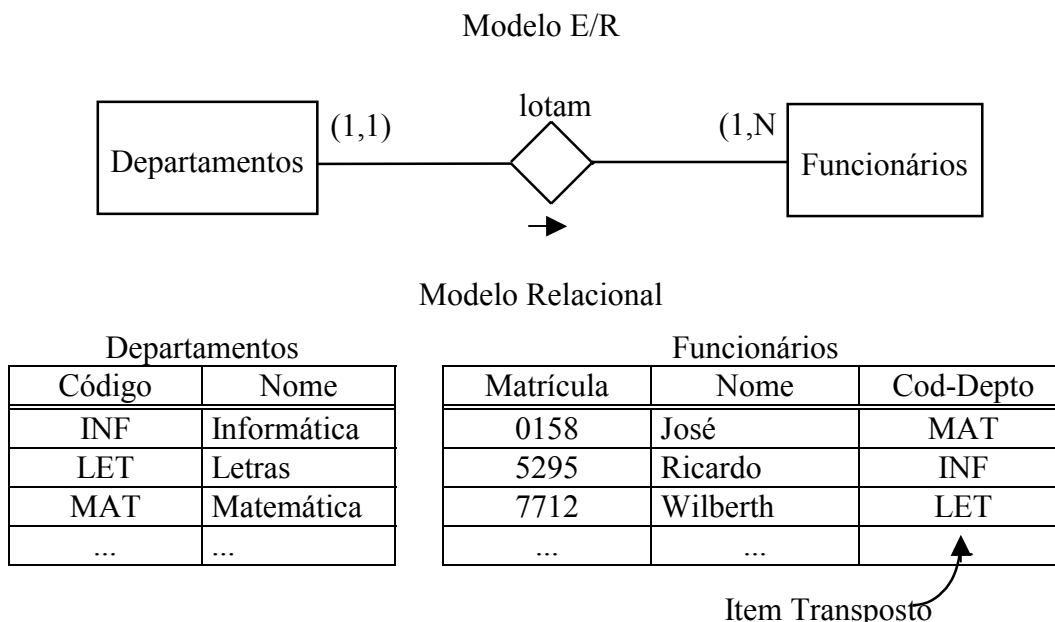


Figura 4.2 - Exemplo de uma ligação entre tabelas, através da transposição de chaves.

Propriedades do Modelo Relacional

- Nenhum campo componente de uma chave primária pode ser nulo;
- Cada célula de uma relação pode ser vazia (exceto de uma chave primária), ou ao contrário, conter no máximo um único valor;
- A ordem das linhas é irrelevante;
- Não há duas linhas iguais;
- Cada coluna tem um nome e colunas distintas devem ter nomes distintos;
- Usando-se os nomes para se fazer referência às colunas, a ordem destas torna-se irrelevante;
- Cada relação recebe um nome próprio distinto do nome de qualquer outra relação da base de dados;
- Os valores de uma coluna são retirados todos de um mesmo conjunto, denominado *domínio* da coluna;
- Duas ou mais colunas distintas podem ser definidas sobre um mesmo domínio;
- Um campo que seja uma chave estrangeira ou um item transposto só pode assumir valor nulo ou um valor para o qual exista um registro na tabela onde ela é chave primária.

4.2 - Diagrama Relacional

Um Diagrama Relacional é a representação gráfica das ligações entre tabelas de um modelo relacional. A figura 4.3 mostra um exemplo de um mapeamento de um diagrama ER para um Diagrama Relacional e suas tabelas correspondentes.

Diagrama E/R

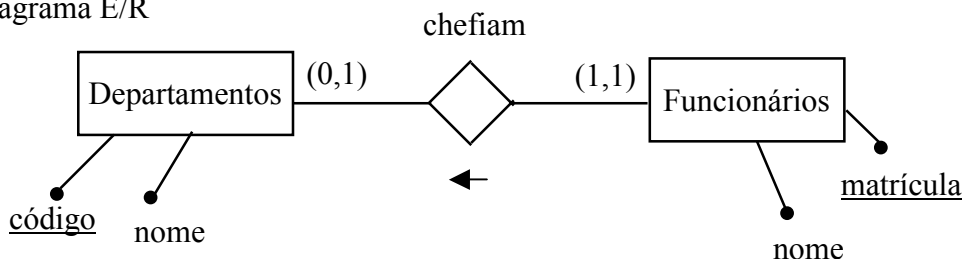
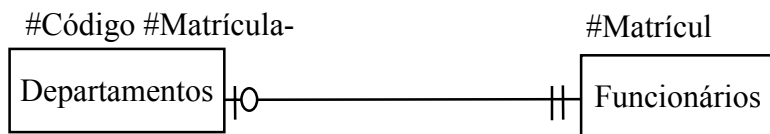


Diagrama Relacional



Tabelas do Modelo Relacional

Departamentos		
Código	Nome	Matrícula-Chefe
INF	Informática	00877
MAT	Matemática	06001
QUI	Química	13888
...

Funcionários	
Matrícula	Nome
13888	Jorge
00877	Dede
06001	Pedro
...	...

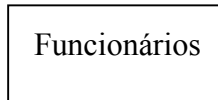
Figura 4.3 - Exemplo de Diagrama Relacional e correspondentes Modelo E/R e Relacional.

Neste exemplo a coluna *Matrícula* foi transposta para a relação *Departamentos*. O contrário também poderia ser feito, isto é, transpor *Código* para *Funcionários*. Escolhemos esta solução porque há poucos funcionários que são gerentes, enquanto todos os departamentos têm gerentes. Assim, a coluna *Matrícula Chefe* não terá valores vazios e dizemos que ela é mais densa do que a coluna resultante da transposição de *Código*.

No Diagrama Relacional são representados os seguintes elementos:

- a) As relações (tabelas) provenientes de conjuntos de entidades e dos agregados do Modelo E/R. São representadas por retângulos, com uma referência à chave primária da tabela em cima.

#Fun

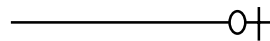


- b) As ligações, que derivam dos relacionamentos, são representadas por linhas contínuas, associadas aos símbolos abaixo:

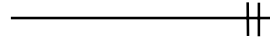
Cardinalidade

Ligação

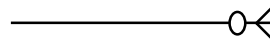
(0,1)



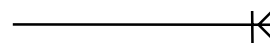
(1,1)



(0,N)



(1,N)



- c) No caso de transposição de chave, se a chave transposta não fizer parte da chave primária da relação destino, iremos representá-la em cima do retângulo desta relação com um subscrito “t”.
- d) Atributos não são representados nos diagramas, mas sim em um dicionário de tabelas do modelo relacional.

Nos capítulos que se seguem, serão discutidos os mapeamentos de um Modelo de Classes no paradigma Orientado a Objetos e de um Modelo de Entidades e Relacionamentos no paradigma Estruturado, para um Modelo Relacional.

5. Projeto Orientado a Objetos

A Análise Orientada a Objetos identifica e define classes que refletem diretamente o domínio do problema e as responsabilidades do sistema dentro dele. O Projeto Orientado a Objetos (*Object-Oriented Design - OOD*) transforma o modelo de análise em um modelo de projeto que serve de base para a construção do software.

Ao contrário dos métodos de projeto de software convencionais, o projeto orientado a objetos resulta em um projeto que obtém um número de diferentes níveis de modularidade. Os componentes principais do sistema são organizados em “módulos” de nível de sistema, chamados subsistemas ou pacotes. Os dados e as operações que os manipulam são encapsulados em classes. Além disso, o OOD deve descrever a estrutura de dados específica dos atributos e os detalhes procedurais de operações individuais.

A análise geralmente transcorre com a suposição de que há uma tecnologia “perfeita” disponível; já no projeto, sabe-se que o sistema será implementado em uma plataforma de hardware, sob um sistema operacional, usando uma linguagem de programação. Em suma, a análise interessa-se pelo *o que* o sistema deve fazer, enquanto o projeto diz respeito a *como* os requisitos serão implementados e, portanto, pressupõe uma infra-estrutura de implementação e é fortemente influenciado por ela. Assim, o projeto de um sistema orientado a objetos é dependente de aspectos como:

- características da *linguagem de programação* a ser utilizada: Qual o tipo de linguagem a ser usada (orientada a objetos, baseada em objetos, convencional,...)? Se é uma LPOO, qual o nível de herança suportado (simples, múltipla)? Quais os mecanismos de acesso a atributos e operações?, etc.
- modelo de *persistência de objetos*: Que mecanismo de persistência de objetos será utilizado (banco de dados orientado a objetos, banco de dados relacional, arquivos, persistência da própria linguagem de programação)?
- características da *plataforma de implementação*: A plataforma de hardware é multi-processada? O sistema é distribuído? etc...
- características da *interface com o usuário*: Que tipo de interface com o usuário será utilizada (interfaces gráficas, orientadas a caracter, ...)?

O Projeto Orientado a Objetos identifica e define classes adicionais, refletindo requisitos de implementação. Deste modo, as ferramentas de modelagem utilizadas na fase de análise - Diagrama de Classes, Diagramas de Interação e Diagramas de Transição de Estados - são utilizadas também na fase de projeto, agora como o intuito de capturar os requisitos de implementação. Entretanto, a perspectiva de implementação existente no projeto, tipicamente, demanda extensões à notação da análise para permitir representar visibilidade, persistência, concorrência, exceções e restrições. Assim, ainda que a UML não especifique explicitamente que notações destes diagramas devem ser utilizadas nas fases de análise e de projeto, algumas de suas facilidades devem ser utilizadas apenas na fase de projeto, tais como as notações de visibilidade de atributos e operações e navegabilidade de relacionamentos.

Ainda que a terminologia e os passos do processo de projeto sejam diferentes para cada um dos métodos de projeto orientado a objetos, os processos globais de projeto são razoavelmente consistentes (Pressman, 2002). De modo geral, dois grandes passos podem ser identificados:

- Projeto da Arquitetura do Sistema: descreve cada um dos subsistemas, de um modo passível de implementação, e as comunicações entre eles;
- Projeto de Objetos: descreve aspectos de implementação de cada uma das classes do sistema, incluindo, o projeto procedural de cada operação, a definição de classes internas e o projeto de estruturas de dados para os atributos das classes.

5.1 - Projeto da Arquitetura do Sistema

A primeira tarefa a ser realizada no OOD consiste em definir uma arquitetura para a aplicação. Será sobre esta arquitetura que o projetista poderá introduzir os aspectos de implementação em um modelo de análise. Em sistemas complexos, a definição da arquitetura do software deve ser iniciada durante a fase de análise para permitir uma melhor organização dos modelos de análise, evitando a complexidade.

A arquitetura de um software deve indicar como o sistema irá funcionar em um ambiente operacional e fornecer os meios necessários para a definição dos componentes do software, das interações entre eles e dos padrões necessários para que todo esse ambiente coopere para produzir o software que está sendo projetado (Magela, 1998).

Uma boa arquitetura de software pode ser obtida através da aplicação de duas idéias fundamentais (Magela, 1998):

- Produção de software em camadas com níveis de abstração definidos; e
- Separação entre interface e implementação.

Desta forma, uma boa arquitetura de software deve ser elaborada em termos dos componentes que compõem este software. Dificilmente é possível compreender o projeto de um software como um todo. Ao contrário, é melhor dividi-lo em componentes gerenciáveis, de complexidade reduzida. Esta quebra deve ser refletida no projeto da arquitetura do sistema. Além disso, a arquitetura deve levar em consideração requisitos tecnológicos ou não-funcionais, tais como segurança, desempenho, manutenibilidade e economia (Magela, 1998).

A organização de classes em pacotes deve ser o ponto de partida para a definição da arquitetura, já que é um meio de estabelecer níveis de abstração para o modelo. Esses níveis de abstração podem ser organizados em camadas e, assim, tratados separadamente durante a fase de projeto. A organização de classes em pacotes é útil também para permitir a produção de componentes para reuso.

Uma forma bastante utilizada para organização em pacotes consiste em agrupar classes de acordo com o tipo de função que exercem no sistema, isto é, o seu estereótipo. Uma classe possui somente um estereótipo. Não podemos ter uma classe com dois

estereótipos, uma vez que ela possuiria duas responsabilidades distintas, o que deveria levar a duas classes distintas (Magela, 1998).

Além da organização por estereótipos, pode ser útil agrupar classes em função do domínio do problema. A organização por domínio de problema conduz à construção de pacotes verticais, levando à produção de componentes de negócio (Magela, 1998). Contudo, esta abordagem isolada é, normalmente, insuficiente. Assim, uma vez que um pacote pode conter outros pacotes, uma abordagem mais eficiente, sobretudo para sistemas complexos, consiste em realizar primeiramente uma organização por domínio do problema e, posteriormente, fazer uma subdivisão dos pacotes do domínio em estereótipos.

Quando esta abordagem baseada no domínio do problema for adotada, o primeiro passo a ser dado consiste em particionar o modelo de análise para definir coleções coesas de classes, relacionamentos e comportamento, empacotando-os em *pacotes* ou *subsistemas*. De fato, este passo é uma revisão da identificação de subsistemas feita na fase de análise, agora levando em conta requisitos de implementação. Subsistemas devem ser definidos e projetados em conformidade com os seguintes critérios (Pressman, 2002):

- Um subsistema deve possuir uma interface bem definida através da qual toda comunicação com o restante do sistema ocorre.
- Com exceção de um pequeno número de “classes de comunicação”, as classes dentro de um subsistema devem colaborar apenas com outras classes deste mesmo subsistema.
- O número de subsistemas deve ser mantido pequeno.

Subsistemas podem ser particionados internamente para auxiliar a reduzir a complexidade. Esta subdivisão poderá ser feita ainda segundo o critério domínio do problema (para problemas muito complexos) ou usando o critério de agrupamento por estereótipos.

5.1.1 – Componentes de Projeto

A comunidade de Smalltalk desenvolveu uma metáfora simples, mas elegante, para uma arquitetura de projeto, conhecida como “*Modelo-Visão-Controlador*” (Model-View-Controller) - MCV. Essencialmente, ela sugere que uma arquitetura típica de projeto orientado a objetos possui três componentes principais: um grupo de classes que *modela* a aplicação em si; um grupo de classes que provê uma *visão* da interface com os usuários; e um grupo de classes que *controla*, ou sincroniza, o comportamento dos demais.

A arquitetura MCV é um bom ponto de partida. Contudo, ela desconsidera um importante componente: a gerência de dados. Isto se deve ao fato de, em Smalltalk, todos os objetos serem naturalmente persistentes. Assim, durante o projeto da arquitetura do sistema, um engenheiro de software deve considerar quatro (e não apenas três) componentes básicos, como mostra a figura 5.1 (Coad et al., 1993):

- *Componente do Domínio do Problema*: corresponde aos subsistemas responsáveis por implementar diretamente os requisitos dos usuários;

- *Componente de Interação Humana*: corresponde aos subsistemas que implementam as interfaces com o usuário;
- *Componente de Gerência de Tarefa*: corresponde aos subsistemas responsáveis por controlar e coordenar tarefas;
- *Componente de Gerência de Dados*: corresponde aos subsistemas responsáveis pelo armazenamento e recuperação de objetos (persistência dos objetos).

Componente de Domínio do Problema (CDP)	Componente de Interação Humana (CIH)	Componente de Gerência de Tarefa (CGT)	Componente de Gerência de Dados (CGD)
--	---	---	--

Figura 5.1: Arquitetura Básica de Projeto Orientado a Objetos (Coad et al., 1993).

A idéia básica dessa arquitetura é simples, mas crucial: ela vai buscar as mesmas classes que foram documentadas no modelo de análise e, então, as envolve com classes adicionais para tratar aspectos relacionados à implementação de gerência de tarefa, gerência de dados e interação humana. Essa arquitetura não só preserva o modelo de análise, como também o utiliza como o cerne do modelo de projeto.

Deve-se notar que, por detrás dessa arquitetura, há uma filosofia de projeto que sugere que as classes centrais, orientadas à aplicação na CDP, não devem estar cientes do “mundo exterior” e não têm de saber como interagir com tal mundo. Este é um ponto fundamental, já que, sem uma atenção consciente a esta filosofia, podemos chegar a uma arquitetura na qual cada classe: (i) sabe como interagir com o usuário final e (ii) sabe como ler e escrever seus dados permanentes em arquivos de disco. Uma abordagem assim poderia funcionar (quem sabe até de maneira mais rápida), mas seria muito suscetível a mudanças na interface com o usuário ou no modelo de persistência. Além disso, fatalmente tornaria a estrutura interna das classes mais complexa do que se tivessem de estar cientes apenas de seus detalhes essenciais, ligados ao domínio da aplicação.

Assim, seguindo a arquitetura básica proposta por Coad e Yourdon (Coad et al., 1993), temos quatro estereótipos:

- Domínio do Problema
- Interface com o Usuário
- Gerência de Tarefas
- Gerência de Dados

De fato, outros estereótipos podem ser usados para organizar classes, tais como Classes Limítrofes, Utilitários, Classes de Exceções, Controladores, etc.

5.2 – Projeto da Componente do Domínio do Problema

No OOD, os resultados da OOA fazem parte da Componente de Domínio do Problema (CDP). Em muitos casos, o modelo de análise desenvolvido pode ser transposto para dentro da CDP sem qualquer alteração adicional. Mas, algumas vezes, ele pode ser modificado ou estendido, de acordo com as necessidades do projeto. Alterações podem advir da necessidade de:

- *reutilizar projetos anteriores e classes já programadas*: uma vez que uma das principais motivações da orientação a objetos é a reutilização de software, é importante que na fase de projeto seja levada em conta a existência de bibliotecas de classes passíveis de serem reusadas. Tipicamente, ajustes feitos para incorporar tais classes envolvem alterações nas hierarquias de generalização-especialização do modelo, de modo a tratar as classes apropriadas da OOA como subclasses de classes de biblioteca pré-existentes, obtendo a vantagem total da habilidade de herdar atributos e métodos de tais classes.

- *ajustar o modelo ao nível de herança suportado*: em função do nível de herança suportado pela linguagem de programação a ser usada na implementação, o modelo da OOA pode ter de ser alterado. Se, por exemplo, o modelo da OOA envolve herança múltipla e a linguagem de implementação não oferece tal recurso, alterações no modelo serão necessárias.

- *ajustar o modelo para melhorar o desempenho*: desempenho pode ser uma preocupação se há um alto tráfego de mensagens entre objetos, se a linguagem de programação implementa herança ineficientemente, ou por várias outras razões advindas da abordagem orientada a objetos. Nestes casos, o projetista pode alterar o modelo de análise para melhor acomodar os ajustes necessários.

- *ajustar o modelo para facilitar o projeto de interfaces com o usuário amigáveis*: com o objetivo de incorporar a característica de qualidade facilidade de uso, pode ser importante considerar novas classes que facilitem a apresentação de listas para seleção do usuário.

A CDP pode ser alterado, ainda, para comportar outros requisitos tecnológicos, tais como segurança, confiabilidade, etc.

5.3 – Projeto da Componente de Interface com o Usuário

A premissa fundamental que justifica a existência deste componente é a seguinte: a porção do sistema que lida com a interface com o usuário deve ser mantida tão independente e separada do resto da arquitetura do software quanto possível. A razão para tal é evidente: aspectos de interface com o usuário provavelmente serão alvo de alterações ao longo de toda a vida produtiva do sistema, e essas alterações devem ter um impacto mínimo (idealmente, *nenhum* impacto) nas partes específicas da aplicação do sistema.

A Componente de Interface com o Usuário (CIU) adiciona detalhes sobre o projeto da interação humana, incluindo formato de janelas e relatórios, entre outros. Nesta etapa, a prototipação é geralmente utilizada, buscando auxiliar o desenvolvimento e a seleção dos

mecanismos reais de interação. A CIU capta como um usuário comandará o sistema e como o sistema apresentará as informações a ele.

Como citado no Capítulo 3, o princípio básico para o projeto de interfaces com o usuário é o seguinte: “Conheça o usuário e as tarefas”. Assim sendo, o modelo de casos de uso deve ser o guia para esta atividade, já que modela exatamente a interação entre atores (classes de usuários) e casos de uso (tarefas ou funcionalidades do sistema).

No desenvolvimento orientado a objetos, o ponto de partida para o projeto da CIU é o modelo de casos de uso e seus cenários e descrições de atores. Com base nos casos de uso, devemos projetar uma hierarquia de comandos, definindo barras de menu, menus *pull-down*, ícones, etc., que levem a ações quando acionados pelo usuário. A hierarquia de comandos deve respeitar convenções e estilos existentes com os quais o usuário já esteja familiarizado. Note que a hierarquia de comandos é, de fato, um meio de apresentar ao usuário as várias funcionalidades disponíveis no sistema, ou, olhando sob outro ponto de vista, as várias mensagens que o usuário pode enviar para as classes dentro do sistema. A hierarquia de comandos deve ser refinada até que todos os casos de uso possam ser implementados, através da navegação nesta hierarquia.

Uma vez definida a hierarquia de comandos, as interações detalhadas entre o usuário e o sistema devem ser projetadas. Neste momento, observe o uso de termos, passos e ações consistentes. Observe, também, aspectos relacionados a desempenho, tempo de resposta e facilidade de uso e aprendizagem. Não obrigue o usuário a ter de lembrar “coisas”. Forneça ajuda interativa.

Normalmente, não é necessário projetar as classes básicas de interfaces gráficas com o usuário. Existem vários ambientes de desenvolvimento de interfaces, oferecendo classes reutilizáveis (janelas, ícones, botões, ...) e, portanto, basta especializar as classes e instanciar os objetos que possuem as características apropriadas para o problema em questão. Em ambientes com interfaces gráficas, a hierarquia de classes básica para a CIU terá tipicamente uma superclasse “janela” e as janelas da aplicação serão construídas adicionando os outros objetos gráficos necessários, tais como botões, menus, ícones.

5.4 – Projeto da Componente de Gerência de Tarefas

A Componente Gerência de Tarefas (CGT) compreende a definição de tarefas e a comunicação e coordenação entre elas. O objetivo básico desta etapa do projeto é definir e classificar as tarefas.

Algumas funcionalidades não são facilmente distribuídas nas classes da CDP, principalmente aquelas que operam sobre vários objetos. Uma possibilidade para resolver tal problema é pulverizar esse comportamento ao longo de vários objetos da CDP ou da CIU. Contudo, esta não é uma boa solução segundo uma perspectiva de alterabilidade. Uma alteração em tal funcionalidade poderia afetar diversos objetos, e assim ser difícil de ser incorporada.

Uma abordagem mais interessante é modelar as classes da CGT como gerenciadores ou coordenadores de tarefas, responsáveis pela realização de tarefas sobre um determinado

conjunto de objetos. Tipicamente, esses gerenciadores agem como aglutinadores, unindo outros objetos para dar forma a um caso de uso. Conseqüentemente, gerenciadores de tarefa são normalmente encontrados diretamente a partir dos casos de uso.

Os tipos de funcionalidade tipicamente atribuídos a gerenciadores de tarefa incluem: comportamento relacionado a transações, seqüências de controle específicas a um caso de uso, ou funcionalidades que separam objetos da CDP e da CIU.

É importante observar, ainda, que os aspectos dinâmicos de um modelo de análise mostram a existência, ou não, de concorrência entre objetos (ou subsistemas). Se objetos (ou subsistemas) não têm de estar ativos em um mesmo momento, então não há necessidade de processamento concorrente e, portanto, o processamento do sistema pode ser atribuído a um único processador. Caso contrário, duas opções de alocação devem ser consideradas (Pressman, 2002):

- alocar cada subsistema a um processador independente: esta abordagem requer sistemas multi-processados ou distribuídos;
- alocar os subsistemas para o mesmo processador e prover suporte a concorrência através de características de sistemas operacionais: neste caso, serão necessárias novas classes de gerência de tarefas, responsáveis por este suporte.

Em um esboço preliminar, pode-se atribuir um gerenciador de tarefa para cada caso de uso, sendo que os seus cenários dão origem a operações da classe que representa o caso de uso. Nesta abordagem, a alterabilidade é facilitada, uma vez que, detectado um problema em um caso de uso, é fácil identificar a classe que trata do mesmo. Um possível problema, contudo, é o desempenho: para sistemas grandes, com muitos casos de uso, haverá muitas classes de gerência de tarefa e, para realizar uma tarefa, pode ser necessária muita comunicação entre essas classes.

Uma solução diametralmente oposta consiste em definir uma única classe de aplicação para todo o sistema. Neste caso, os cenários de todos os casos de uso dão origem a operações dessa classe. Fica evidente que, exceto para sistemas muito pequenos, a classe de aplicação será extremamente complexa e, portanto, esta abordagem não seria prática.

Normalmente, uma solução intermediária entre as duas anteriormente apresentadas conduz a melhores resultados. Nesta abordagem, casos de uso complexos são designados a classes de gerência de tarefas específicas. Casos de uso mais simples e de alguma forma relacionados são tratados por uma mesma classe de aplicação.

Uma coisa é certa: pelo menos um gerenciador de tarefa será sempre necessário - a classe Aplicação, representando o sistema como um todo. Os objetos desta classe representam as várias sessões (execuções) do sistema.

Obviamente, é necessário levar em conta, ainda, quantos executáveis devem ser gerados para o sistema. Se mais do que um for necessário, cada executável terá de dar origem a uma classe de aplicação. Outros fatores que afetam esta decisão são aspectos de distribuição geográfica e se o sistema será um aplicativo ou um sistema rodado a partir de um navegador.

A hierarquia de tarefas a serem realizadas oferece um recurso bastante útil para a definição das janelas, menus ou outros componentes de interação necessários para cada uma dessas tarefas. Assim os projetos das componentes de interação humana e de gerência de tarefa estão bastante relacionados, e devem ser realizados conjuntamente, uma vez que, muitas vezes, são as tarefas que determinam a necessidade de elementos de interface com o usuário para sua execução.

5.5 – Projeto da Componente de Gerência de Dados

A Componente de Gerência de Dados (CGD) provê a infra-estrutura básica para o armazenamento e a recuperação de objetos no sistema. Sua finalidade é isolar os impactos da tecnologia de gerenciamento de dados sobre a arquitetura do software (Coad et al., 1993). A questão primordial neste momento do projeto é: Como tornar persistentes os objetos do sistema?

De modo geral, são quatro as opções atualmente disponíveis para suportar a persistência de objetos:

- uso de arquivos: Neste caso, é necessário estabelecer uma estratégia para escrita e leitura de uma série de objetos em um arquivo simples. Um layout simples para um arquivo pode ser pensado em termos de um objeto por linha, com os atributos do objeto iniciando e terminando em posições específicas. As facilidades oferecidas pelas próprias linguagens de programação para manipular arquivos devem ser utilizadas.
- persistência de objetos fornecida pela própria linguagem de programação, como é o caso de Smalltalk (arquivo IMAGE) e Eiffel (classe STORABLE). Em Smalltalk, todos os objetos são persistentes e, ao encerrar uma sessão, o estado de todo ambiente é salvo em um arquivo. Neste caso, não há projeto da CGD. Eiffel, por sua vez, oferece a classe STORABLE, que oferece mecanismos para salvar e recuperar objetos persistentes. Neste caso, o projeto da CGD consiste em definir que classes devem ser definidas como sub-classes de STORABLE.
- bancos de dados de objetos: Em um ambiente orientado a objetos, a solução ideal para essa questão seria usar um Sistema de Gerenciamento de Bancos de Dados Orientado a Objetos (SGBDOO), onde cada uma das classes persistentes na arquitetura de software corresponderia a exatamente uma base de dados gerenciada pelo SGBDOO.
- bancos de dados relacionais: Ainda que haja diferenças entre a abordagem relacional e o paradigma de objetos, os bancos de dados relacionais tem sido utilizados pela maioria dos desenvolvedores OO para armazenar objetos (Ambler, 1998). Alguns ambientes de programação, tais como certos ambientes C++ e o ambiente Delphi (Object Pascal), oferecem facilidades de interface com alguns bancos de dados relacionais. Neste caso, o projeto da CGD é fortemente dependente do nível de suporte oferecido. Caso o ambiente encapsule totalmente o banco de dados relacional, o projeto pode ser muito semelhante ao projeto

usando um banco de dados OO; caso contrário, o projetista deve efetuar um projeto de bases de dados relacionais, definindo suas tabelas, para só então poder utilizá-las no projeto OO da CGD. Utilizando SGBDs relacionais, geralmente, o processo de projeto começa pela tradução das classes no modelo orientado a objetos para a terceira forma normal padrão. Para cada tabela em terceira forma normal, derivada deste processo de “normalização de objetos”, uma tabela na base de dados é definida.

A despeito da opção de persistência adotada, outra questão deve ser considerada: Que classes devem suportar a persistência dos objetos?

Uma alternativa seria tornar cada classe, ao longo de toda a arquitetura do software, responsável por suas próprias atividades de leitura e escrita. Obviamente, nesta abordagem, a arquitetura torna-se completamente dependente da tecnologia de persistência e, se, por exemplo, a organização migra de um sistema de arquivo para um SGBD relacional, ou mesmo de um SGBD relacional para outro, essa migração impactaria todas as classes ao longo do sistema.

Uma abordagem mais elegante consiste em fazer com que apenas uma parte da arquitetura de software fique ciente da tecnologia de persistência adotada. Essa parte, a CGD, serve como uma camada intermediária para as classes de objetos persistentes, tipicamente da CDP, estabelecendo um protocolo para a persistência dos objetos. Via conexões de mensagem, a CGD lê e escreve dados, estabelecendo uma comunicação entre a base de dados e os objetos do sistema. O preço a ser pago por este tipo de “ocultamento de informação” é o desempenho: cada requisição para ler ou escrever um objeto envolve não apenas os comandos físicos de leitura/gravação, mas também uma troca de dados (via parâmetros de mensagem) entre a CGD e o objeto no sistema (Yourdon, 1994). Nesta abordagem, as operações de armazenamento e recuperação de objetos não são colocadas nas classes da CDP, mas sim em uma ou mais classes “salvadoras de objetos” dentro da CGD.

A abordagem mais direta para esta camada de persistência consiste em prover uma classe “sombra” na CGD para cada classe persistente nos demais componentes da arquitetura. Tal classe salvadora de objetos encapsula a funcionalidade necessária para se implementar a persistência dos objetos da classe correspondente.

Uma abordagem mais elegante e complexa consiste em prover classes genéricas que estabelecem protocolos para comunicação com os meios de armazenamento secundários e utilizá-las para a persistência dos objetos das classes correspondentes.

Uma vez que os bancos de dados relacionais são os dispositivos de armazenamento mais confiáveis e utilizados atualmente (Magela, 1998), a seguir detalhamos o projeto da CGD pressupondo o uso deste dispositivo de armazenamento.

5.5.1 - Persistência em Bancos de Dados Relacionais

Claramente, há uma diferença semântica significativa entre o modelo de classes de um projeto orientado a objetos e o modelo relacional. Assim, para que a persistência de

objetos seja feita em um banco de dados relacional, é necessário proceder um mapeamento entre esses dois mundos. Deve-se realçar que este mapeamento só deve ser visível na camada de persistência, isto é, na CGD, isolando a CDP do impacto da tecnologia de bancos de dados. No mapeamento dos mundos de objetos e relacional, as seguintes questões devem ser abordadas:

- Mapeamento de Classes para Tabelas e de Objetos para Linhas;
- Mapeamento de Herança.
- Mapeamento de Associações entre Objetos;

Mapeando Classes em Tabelas e Objetos em Linhas

Quando não há herança, cada classe deve ser mapeada em uma tabela e cada instância da classe (objeto) em uma linha desta tabela. O modelo de classes deve ser normalizado previamente para a 3ª forma normal, eliminando-se atributos multivalorados. Neste processo de normalização das classes, surge uma importante questão. No modelo relacional, toda tabela tem de ter uma chave primária, isto é, uma ou mais colunas, cujos valores identificam univocamente um registro da mesma. Objetos, por sua vez, têm identidade própria, independentemente dos valores de seus atributos. Assim, que identificador único devemos designar aos nossos objetos no banco de dados relacional, para que possamos distingui-lo? Uma solução possível consiste em observar se há um atributo na classe com esta propriedade de identificação única e utilizá-lo, então, como chave primária. Caso não haja um atributo com tal característica, deverá ser criado um. Esta abordagem deve ser utilizada sempre que já houver uma base de dados legada, sendo utilizada por outros sistemas, não orientados a objetos.

Uma maneira mais eficaz, sobretudo para permitir a construção de componentes mais genéricos de persistência, consiste em dar a cada objeto um atributo chamado de identificador de objeto (IDO). Os IDOs são tipicamente implementados como grandes números inteiros, que são utilizados como chaves primárias nas tabelas do banco de dados relacional. Essa coluna na tabela do banco de dados relacional deverá ser do tipo auto-incremento, garantindo que a unicidade do objeto será mapeada na tabela, e não deve possuir nenhum significado de negócio (Ambler, 1998) (Magela, 1998).

Mapeando Herança

A grande questão no mapeamento da herança diz respeito a como organizar os atributos herdados no banco de dados. Existem três soluções razoavelmente aplicáveis para mapear a herança em um banco de dados relacional (Ambler, 1998):

1. Utilizar uma tabela para toda a hierarquia;
2. Utilizar uma tabela por classe concreta na hierarquia;
3. Utilizar uma tabela por classe na hierarquia.

No primeiro caso, a tabela derivada contém os atributos de todas as classes na hierarquia. A vantagem desta solução é a simplicidade da implementação. Ela suporta bem

o polimorfismo e facilita a designação de IDOs, já que todos os objetos estão em uma única tabela. O problema fundamental desta solução é que, se as subclasses têm muitos atributos diferentes, haverá muitas colunas que não se aplicam aos objetos individualmente, provocando grande desperdício de espaço no banco de dados. Além disso, sempre que um atributo for adicionado a qualquer classe na hierarquia, um novo atributo deve ser adicionado à tabela. Isso aumenta o acoplamento na hierarquia, pois, se um erro for introduzido durante a adição do atributo, todas as classes na hierarquia podem ser afetadas e não apenas a classe que recebeu o novo atributo.

No segundo caso, utiliza-se uma tabela para cada classe concreta na hierarquia. Cada tabela derivada para as classes concretas inclui tanto os atributos da classe quanto os de suas superclasses. A grande vantagem é a facilidade de processamento sobre as subclasses concretas, já que todos os dados de uma classe concreta estão armazenados em uma única tabela. Da mesma forma que o caso anterior, a designação de IDOs é facilitada, com a vantagem de se eliminar o desperdício de espaço. Contudo, há ainda algumas desvantagens. Quando uma superclasse é alterada, por exemplo, é necessário alterar as tabelas de todas as suas subclasses. Além disso, quando há muito processamento envolvendo a superclasse, há uma tendência de queda do desempenho da aplicação, já que passa a ser necessário manipular várias tabelas ao invés de uma.

A terceira solução é a mais genérica: utiliza-se uma tabela por classe, não importando se concreta ou abstrata. Deve haver uma tabela para cada classe e visões para cada uma das classes derivadas (subclasses). De fato, esta abordagem é a que está mais de acordo com os conceitos da orientação a objetos. É muito mais fácil modificar uma superclasse e acrescentar subclasses, já que é necessário apenas alterar ou acrescentar uma tabela. Uma desvantagem é o grande número de tabelas no banco de dados, uma para cada classe. Além disso, pode levar mais tempo para acessar dados de uma classe, uma vez que pode ser necessário acessar várias tabelas.

Mapeando Associações

Uma vez que a persistência será feita em um banco de dados relacional, é necessário transpor chaves entre tabelas para mapear associações. As regras válidas para o modelo relacional têm de ser aplicadas, tal como criar tabelas associativas para mapear associações muitos-para-muitos. Para implementar associações um-para-um ou um-para-muitos, basta transpor a chave primária de uma tabela para a outra.

5.6 - Projeto de Objetos

No contexto do Projeto Orientado a Objetos, devemos desenvolver um projeto detalhado dos atributos, das associações e das operações que compõem cada classe, e uma especificação das mensagens que conectam as classes com seus colaboradores.

Inicialmente, uma descrição de protocolo para cada classe deve ser provida, estabelecendo o conjunto de mensagens da classe (sua interface) e uma descrição da operação a ser executada quando um objeto da classe receber uma dessas mensagens. Neste momento, deve-se definir, portanto, que operações e atributos devem ser públicos ou

privados à classe. A seguir, deve-se fazer uma descrição da implementação da classe, provendo detalhes internos necessários para a implementação, mas não necessários para a comunicação entre objetos.

No que tange aos atributos, esta descrição deve conter uma especificação das estruturas de dados privadas da classe, com indicações de itens de dados e tipos para os atributos. Deve-se definir, também, a navegabilidade das associações. Esta decisão conduzirá à definição de novas variáveis na classe, bem como do seu tipo e estrutura de dados.

Para as operações, deve-se definir os tipos e estruturas de dados para as interfaces, bem como uma especificação procedural de cada operação (projeto algorítmico). No caso de operações complexas, é uma boa opção modularizá-las, criando sub-operações, estas privadas à classe. O projeto algorítmico de uma operação pode revelar a necessidade de variáveis locais aos métodos ou de variáveis globais à classe para tratar detalhes internos.

5.7 - Critérios de Qualidade de Projeto OO

Há vários projetos possíveis que podem implementar corretamente um conjunto de requisitos. Um bom projeto equilibra custo e benefício de modo a minimizar o custo total do sistema ao longo de seu tempo de vida total. Coad e Yourdon propõem alguns critérios baseados na observação e estudos de casos reais de desenvolvimento OO, entre eles (Coad et al., 1993):

- *Acoplamento*: diz respeito ao grau de interdependência entre componentes de software. O objetivo é minimizar o acoplamento, isto é, tornar os componentes tão independentes quanto possível. No OOD, estamos preocupados principalmente com o acoplamento entre classes e entre subsistemas. A meta é minimizar o número de mensagens trocadas e a complexidade e o volume de informação nas mensagens.
- *Coesão*: define como as atividades de diferentes componentes de software estão relacionadas umas com as outras. Vale a pena ressaltar que coesão e acoplamento são interdependentes e, portanto, uma boa coesão, geralmente, conduz a um pequeno acoplamento. No OOD, três níveis de coesão devem ser verificados:
 - ⇒ *coesão de métodos individuais*: um método deve executar uma e somente uma função;
 - ⇒ *coesão de classes*: atributos e serviços encapsulados em uma classe devem ser altamente coesos, isto é, devem estar estreitamente relacionados; e
 - ⇒ *coesão de uma hierarquia de classes*: a coesão de uma hierarquia pode ser avaliada examinando-se até que extensão uma subclasse redefine ou cancela atributos e métodos herdados da superclasse.
- *Clareza*: um projeto deve ser passível de entendimento por outros projetistas.
- *Reutilização*: bons projetos devem ser fáceis de serem reutilizados.

- *Efetivo Uso da Herança*: para sistemas médios, com aproximadamente 100 classes, as hierarquias devem ter de 2 a 7 níveis de generalização-especialização. Projetos com uso intensivo de herança múltipla devem ser evitados, pois são mais difíceis de serem entendidos e, conseqüentemente, de serem reutilizados e mantidos.
- *Protocolo de Mensagens Simples*: protocolos de mensagem complexos são uma indicação comum de acoplamento excessivo entre classes. Assim, a passagem de muitos parâmetros deve ser evitada.
- *Operações Simples*: os métodos que implementam as operações de uma classe devem ser bastante pequenos. Se um método envolve muito código, é uma forte indicação de que as operações da classe foram pobremente modularizadas.
- *Habilidade de “avaliar por cenário”*: é importante que um projeto possa ser avaliado a partir de um cenário particular escolhido. Revisores devem poder representar o comportamento de classes e objetos individuais, e assim, verificar o comportamento dos objetos nas circunstâncias desejadas.

5.8 – Revisão do Documento de Projeto

Assim como os demais documentos, o documento de projeto deve ser revisado. Os seguintes aspectos devem ser observados:

- Aderência a padrões de documento de projeto;
- Aderência a padrões de nomenclatura, incluindo nomes de classes, atributos, operações, mensagens, etc;
- Coerência com os modelos de análise e de especificação de requisitos.

Do ponto de vista de coerência entre modelos, os seguintes aspectos devem ser observados:

- As classes da componente do domínio do problema devem ser necessárias e suficientes para cumprir as responsabilidades apontadas pelos casos de uso do documento de especificação de requisitos, agora já com uma perspectiva de implementação.
- As classes da componente de interação humana devem ser necessárias e suficientes para permitir o acesso e a realização de todos os casos de uso do documento de especificação de requisitos e seus cenários.
- As classes da componente de gerência de tarefas devem controlar todos os casos de uso documentados na especificação de requisitos e seus cenários;
- As classes da gerência de dados devem ser necessárias e suficientes para tratar do armazenamento e recuperação de objetos de todas as classes persistentes do sistema (tipicamente, as classes da componente do domínio do problema);

- Alterações não decorrentes da tecnologia, mas da detecção de um erro de especificação de requisitos ou de análise, devem ser atualizadas nos correspondentes modelos de especificação de requisitos ou análise.

5.9 – Padrões de Projeto (*Design Patterns*)

Projetar software é uma tarefa difícil. A maioria dos projetistas é capaz de compreender conceitos como classes, objetos, interfaces e herança. O desafio reside em aplicá-los para construir software flexível e reutilizável. Projetistas novatos tendem a se perder com as muitas opções disponíveis e a recorrer a técnicas não orientadas a objetos com as quais têm certa familiaridade. Projetistas experientes, por sua vez, tendem a fazer bons projetos. Eles sabem que não se deve resolver todo e qualquer problema partindo de princípios básicos (classes, objetos, herança, relacionamentos, agregação,...). Ao contrário, deve-se buscar reutilizar soluções que funcionaram no passado. Esta é a motivação para o estudo de padrões de projeto, ou *design patterns*.

Assim, um padrão é um par nomeado problema/solução, que pode ser utilizado em novos contextos, com orientações sobre como utilizá-lo em novas situações [Larman00]. O objetivo de um *design pattern* é registrar uma experiência no projeto de software OO, na forma de um padrão passível de ser efetivamente utilizado por projetistas. Cada padrão sistematicamente nomeia, explica e avalia um importante projeto que ocorre repetidamente em sistemas OO (Gamma et al., 1995).

Um projetista familiarizado com padrões de projeto pode aplicá-los diretamente a problemas de projeto sem ter que redescobrir abstrações e os objetos que as capturam. Uma vez que um padrão é aplicado, muitas decisões de projeto decorrem automaticamente.

Em geral, um padrão tem quatro elementos essenciais:

- **Nome:** identificação de uma ou duas palavras, que se possa utilizar para descrever o problema de projeto, suas soluções e conseqüências.
- **Problema:** descreve quando aplicar o padrão. Explica o problema de projeto e seu contexto.
- **Solução:** descreve os elementos que compõem o projeto, seus relacionamentos, responsabilidades e colaborações. **Não** descreve um particular projeto concreto ou implementação. Um padrão provê uma descrição abstrata de um problema de projeto e como uma organização geral de classes e objetos resolve este problema.
- **Conseqüências:** são os resultados e os comprometimentos feitos ao se aplicar o padrão, tais como, bibliotecas de classes necessárias, que problemas acarreta, etc.

5.9.1 – O Catálogo proposto por (Gamma et al., 1995)

Um dos principais catálogos de padrões OO publicados até o momento é o apresentado em (Gamma et al., 1995). A descrição dos padrões neste catálogo é mais detalhada e consiste de:

- **Nome:** nome dado ao padrão neste catálogo.
- **Classificação:** é feita segundo dois critérios: propósito e escopo. O propósito reflete o que o padrão faz, isto é, sua funcionalidade. Segundo este critério, um padrão pode ser: criativo, diz respeito ao processo de criação de objetos; estrutural, lida com a composição de classes ou objetos; ou comportamental, caracteriza os meios pelos quais classes ou objetos interagem e distribuem responsabilidades. O escopo, por sua vez, especifica se o padrão está centrado em classes (e neste caso, faz intenso uso de herança) ou em objetos (e, portanto, mais apoiado em associações).
- **Intenção (Propósito):** descreve sucintamente o que faz o padrão, seu propósito e o problema endereçado.
- **Também conhecido como:** apresenta outros nomes pelos quais o padrão é conhecido (se houver).
- **Motivação:** apresenta um cenário que ilustra o problema endereçado pelo padrão e como a estrutura proposta pelo padrão resolve este problema.
- **Aplicabilidade:** trata de situações nas quais o padrão pode ser aplicado e como reconhecer essas situações.
- **Estrutura:** apresenta o modelo de classes do padrão e, opcionalmente, diagramas de interação para ilustrar seqüências de requisições e colaborações entre objetos.
- **Participantes:** fornece uma descrição das classes e/ou objetos que participam do padrão e suas responsabilidades.
- **Colaborações:** descreve como os participantes colaboram para realizar suas responsabilidades.
- **Conseqüências:** trata dos comprometimentos e resultados quando se aplica o padrão, tanto positivos como negativos.
- **Implementação:** discute armadilhas e sugestões na implementação do padrão, bem como técnicas e questões específicas de linguagem.
- **Código-Exemplo:** apresenta fragmentos de código em C++ ou Smalltalk que ilustram como o padrão pode ser implementado.
- **Usos conhecidos:** apresenta exemplos de uso do padrão encontrados em sistemas reais.

- **Padrões relacionados:** faz referência a outros padrões proximamente relacionados com o padrão em questão, discutindo diferenças. Relaciona, também, outros padrões que devem ser utilizados juntamente com este.

Tendo em vista a classificação proposta por (Gamma et al., 1995), é possível apontar os objetivos gerais de cada grupo de padrões. Quanto ao propósito, os seguintes objetivos são válidos:

- **Padrão Criativo:** abstrai o processo de instanciação (criação) de objetos, ajudando a tornar um sistema independente de como seus objetos são criados, compostos e representados.
 - Padrão Criativo de Classe: utiliza herança para variar a classe instanciada, adiando alguma parte da criação de objetos para subclasses.
 - Padrão Criativo de Objeto: delega a instanciação de um objeto para outro objeto ou adia alguma parte da criação de um objeto para outro objeto.
- **Padrão Estrutural:** diz respeito a como classes e objetos são compostos para formar estruturas maiores.
 - Padrão Estrutural de Classe: utiliza herança para compor classes.
 - Padrão Estrutural de Objeto: descreve meios de compor objetos a partir de outros objetos, visando obter nova funcionalidade. A flexibilidade adicional da composição de objetos advém da habilidade de alterar uma composição em tempo de execução, o que é impossível com a composição estática de classes.
- **Padrão Comportamental:** diz respeito a algoritmos e a atribuição de responsabilidades entre objetos.
 - Padrão Comportamental de Classe: utiliza herança para distribuir comportamento entre classes, ou seja, para descrever algoritmos e fluxos de controle.
 - Padrão Comportamental de Objeto: utiliza composição de objetos para distribuir o comportamento. Descreve como um grupo de objetos coopera para realizar uma tarefa que nenhum objeto poderia realizar sozinho.

A tabela 5.1 apresenta o catálogo de padrões proposto por (Gamma et al., 1995). Na seqüência, é apresentada uma breve descrição de cada um dos padrões.

Escopo	Propósito		
	Criativo	Estrutural	Comportamental
Classe	Método-Fábrica	Adaptador (classe)	Interpretador Método Modelo
Objeto	Construtor Fábrica Abstrata Protótipo Singular	Adaptador (objeto) Composto Decorador Fachada Peso-Mosca Ponte Procurador	Cadeia de Responsabilidade Comando Iterador Mediador Memorial Observador Estado Estratégia Visitador

Tabela 5.1 – Catálogo de Padrões proposto em (Gamma et al., 1995).

Padrões de Classe

- **Método-Fábrica:** define uma interface para a criação de objetos, mas deixa que uma classe adie a instanciação para suas subclasses.
- **Adaptador:** converte a interface de uma classe em outra interface, permitindo que classes trabalhem em conjunto, quando isto não seria possível por causa da incompatibilidade de interfaces.
- **Método Modelo:** define o esqueleto de um algoritmo em uma operação, adiando alguns de seus passos para as subclasses, permitindo que as subclasses redefinam certos passos do algoritmo sem alterar sua estrutura.
- **Interpretador:** Dada uma linguagem, define uma representação para sua gramática, junto com um interpretador que utiliza essa representação para interpretar sentenças na linguagem.

Padrões de Objetos

- **Construtor:** separa a construção de um objeto complexo de sua representação de modo que o mesmo processo de construção pode criar diferentes representações.
- **Fábrica Abstrata:** provê uma interface para a criação de famílias de objetos relacionados ou dependentes, sem especificar suas classes concretas.

- **Protótipo:** especifica os tipos de objetos que podem ser criados a partir de uma instância prototípica e cria novos objetos copiando este protótipo.
- **Singular:** garante que uma classe possui uma única instância e provê um ponteiro global para acessá-la.
- **Composto:** compõe objetos em estruturas de árvore para representar hierarquias todo-parte, permitindo que clientes tratem objetos individuais e compostos uniformemente.
- **Decorator:** anexa responsabilidades adicionais a um objeto dinamicamente, permitindo estender sua funcionalidade.
- **Fachada:** provê uma interface unificada para um conjunto de interfaces em um subsistema. Define uma interface de nível mais alto para o subsistema, tornando-o mais fácil de ser usado.
- **Peso-Mosca:** utiliza compartilhamento para suportar eficientemente um grande número de objetos de granularidade muito fina.
- **Ponte:** desacopla uma abstração de sua implementação, de modo que ambas possam variar independentemente.
- **Procurador:** provê um substituto/procurador (*proxy*) que tem autorização para controlar o acesso a um objeto.
- **Cadeia de Responsabilidades:** evita o acoplamento entre o objeto emissor de uma mensagem e o receptor, dando chance para mais de um objeto tratar a solicitação. Encadeia os objetos receptores e passa a mensagem adiante na cadeia até que um objeto a trate.
- **Comando:** encapsula uma requisição como um objeto, permitindo, assim, parametrizar clientes com diferentes requisições e desfazer operações (comando *undo*).
- **Iterador:** provê um meio de acessar sequencialmente os elementos de um objeto agregado sem expor sua representação básica.
- **Mediador:** define um objeto que encapsula como um conjunto de objetos interage.
- **Memorial:** sem violar o encapsulamento, captura e externaliza o estado interno de um objeto de modo que se possa posteriormente restaurar o objeto para este estado.
- **Observador:** define uma dependência um-para-muitos entre objetos de modo que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.
- **Estado:** permite que um objeto altere o seu comportamento quando seu estado interno muda, fazendo parecer que o objeto mudou de classe.

- **Estratégia:** define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. Deste modo, o algoritmo varia independentemente dos clientes que o utilizam.
- **Visitador:** representa uma operação a ser executada sobre os elementos de uma estrutura de um objeto. Permite definir uma nova operação sem alterar as classes dos elementos sobre as quais ele opera.

Gamma et al. (1995) sugerem que, para se utilizar um padrão do catálogo, os seguintes passos devem ser seguidos:

1. Leia o padrão uma vez para obter uma visão geral, concentrando a atenção nas seções de Aplicabilidade e Consequências para garantir que este é o padrão certo para o seu problema.
2. Volte e estude as seções de Estrutura, Participantes e Colaborações. Tenha a certeza de que compreendeu as classes e objetos no padrão e como se relacionam entre si.
3. Olhe a seção Código de Exemplo para ver um exemplo concreto do padrão em código. Isto irá ajudá-lo a aprender a implementar o padrão.
4. Escolha nomes para os participantes (classes e/ou objetos) do padrão que sejam significativos no contexto da aplicação. Os nomes em um padrão de projeto são geralmente muito abstratos para aparecerem diretamente em uma aplicação. Contudo, é útil incorporar o nome do participante do padrão de projeto ao seu nome na aplicação, de modo a tornar o padrão mais explícito na implementação.
5. Defina as classes.
6. Defina nomes específicos da aplicação para as operações no padrão.
7. Implemente as operações para realizar as responsabilidades e colaborações do padrão.

Padrões de projeto não devem ser aplicados indiscriminadamente. Frequentemente, eles alcançam flexibilidade e variabilidade através da introdução de níveis adicionais de indireção que podem complicar um projeto e/ou resultar em queda de desempenho.

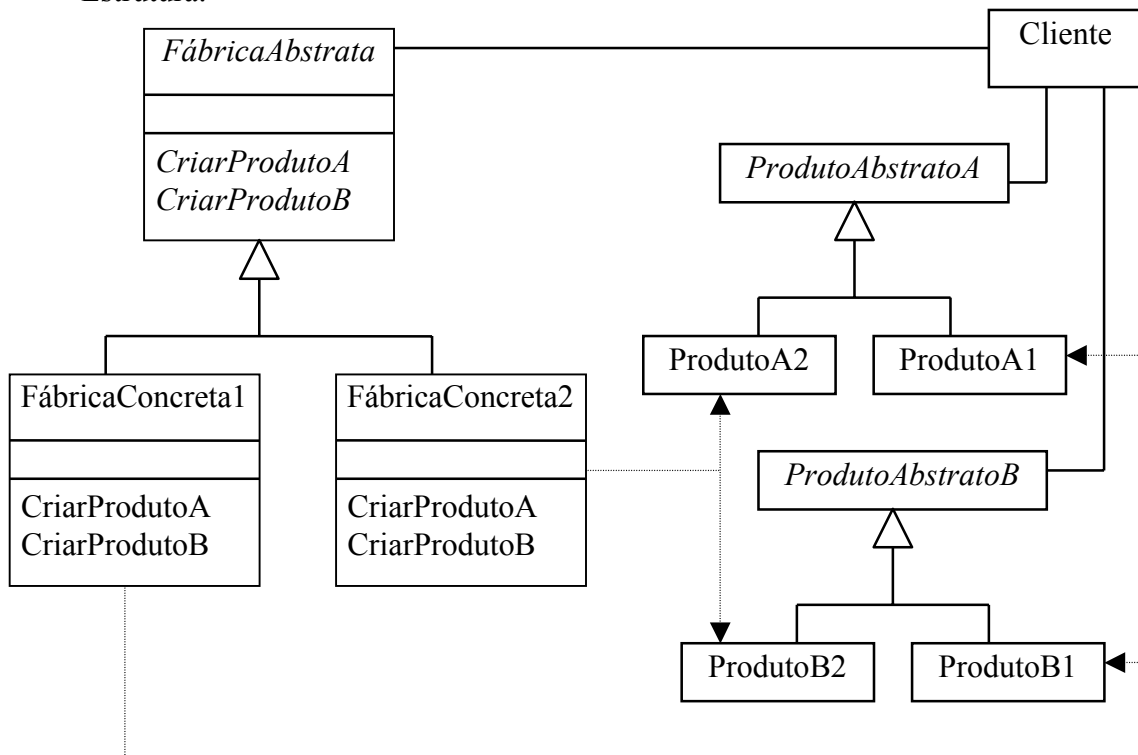
Um padrão de projeto só deve ser aplicado quando a flexibilidade que ele proporciona for realmente necessária. A seção de Consequências é muito útil para a avaliação dos benefícios e obrigações de um padrão.

Padrões de projeto são bastante úteis para a criação de projetos robustos, aptos a suportar determinadas mudanças, garantindo que o sistema pode ser alterado de certas maneiras específicas. Cada padrão permite que algum aspecto da estrutura do sistema varie de forma independente de outros aspectos, tornando o sistema mais robusto para um particular tipo de alteração.

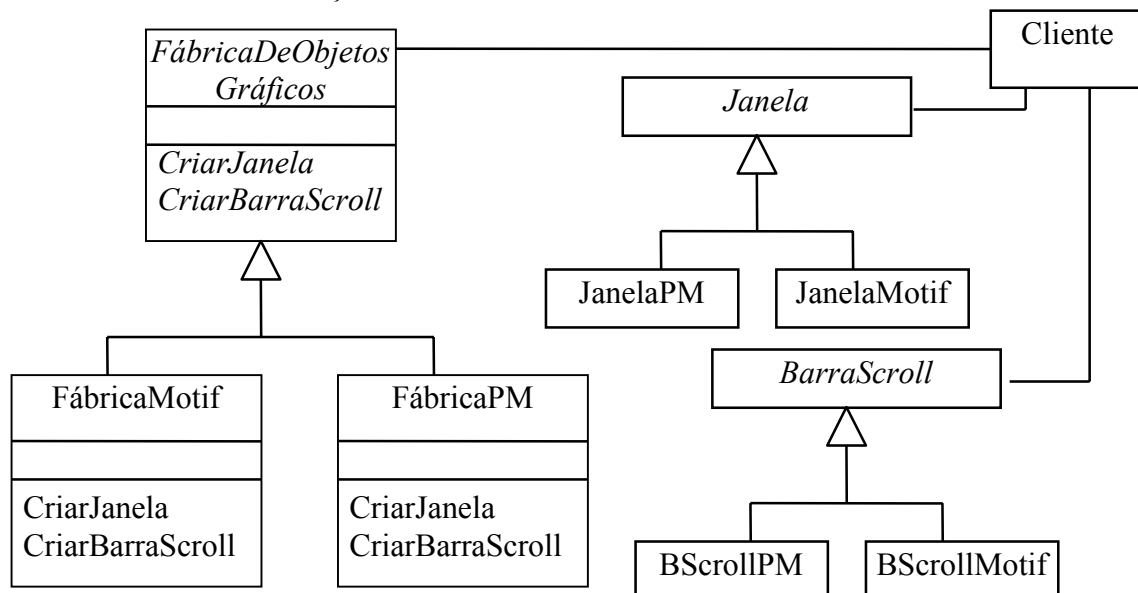
A seguir, são parcialmente apresentados alguns dos padrões de projeto propostos em (Gamma et al., 1995).

Fábrica Abstrata

- Classificação: Padrão Criativo de Objeto
- Propósito: Prover uma interface para criar famílias de objetos relacionados ou dependentes, sem especificar suas classes concretas.
- Também conhecido como: Kit.
- Motivação: Toolkit de Interface Gráfica com o Usuário, suportando diferentes padrões de apresentação (Motif, Presentation Manager,...). Cada padrão de apresentação define diferentes comportamento e aparência para objetos de interface, tais como janelas, botões, barras de scroll, etc. Para ser portátil ao longo de diferentes padrões de apresentação, uma aplicação não pode se comprometer com um padrão específico.
- Aplicabilidade:
 - Sistema deve ser independente de como seus produtos são criados, compostos e representados.
 - Sistema deve ser configurado com uma dentre várias famílias de produtos.
 - Uma família de produtos relacionados foi projetada para ser usada em conjunto e esta restrição tem de ser garantida.
- Estrutura:

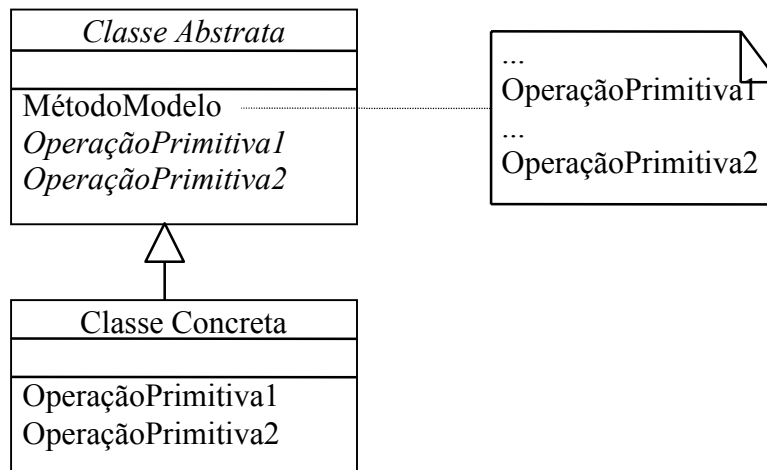


- Participantes:
 - Fábrica Abstrata: declara uma interface para operações criam objetos-produto abstratos;
 - Fábrica Concreta: implementa as operações para criar objetos-produto concretos;
 - Produto Abstrato: declara uma interface para um tipo de objeto produto.
 - Produto Concreto: implementa a interface abstrata de Produto Abstrato e define um objeto-produto a ser criado pela Fábrica Concreta correspondente.
 - Cliente: utiliza apenas as interfaces declaradas por Fábrica Abstrata e Produto Abstrato.
- Colaborações: Fábrica Abstrata adia a criação de objetos-produto para suas subclasses Fábricas Concretas.
- Consequências:
 - Isola classes concretas: uma vez que uma fábrica encapsula a responsabilidade e o processo de criação de objetos-produto, ela isola clientes das classes de implementação.
 - Fica mais fácil a troca de uma família de produtos, bastando trocar a fábrica concreta usada pela aplicação.
 - Promove consistência entre produtos. Quando objetos-produto em uma família são projetados para trabalhar juntos, é importante que uma aplicação utilize apenas objetos desta família.
 - O suporte a novos tipos de produtos é dificultado, já que a interface da Fábrica Abstrata fixa o conjunto de produtos que podem ser criados. Para suportar novos tipos de produtos, é necessário alterar a interface da fábrica, o que envolve alterações na Fábrica Abstrata e em todas as suas subclasses.



Método Modelo

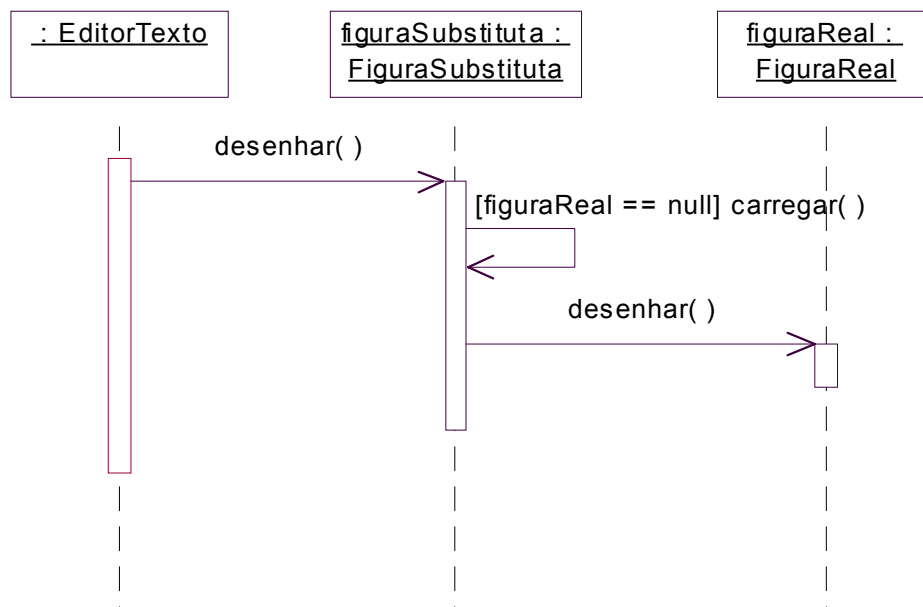
- Classificação: Padrão Comportamental de Classe.
- Propósito: Definir o esqueleto de um algoritmo em uma operação, adiando alguns passos para as subclasses.
- Aplicabilidade: Para implementar partes invariantes de um algoritmo apenas uma vez e deixar a cargo das subclasses a implementação do comportamento variável.
- Estrutura:



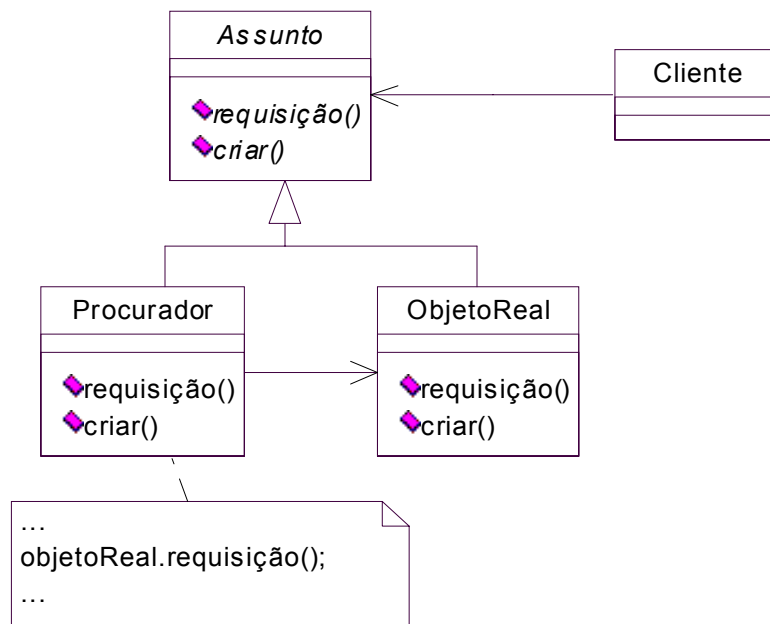
- Participantes:
 - Classe Abstrata: implementa um método modelo, definindo o esqueleto de um algoritmo e define operações primitivas abstratas que as subclasses concretas têm de definir para implementar os passos do algoritmo;
 - Classe Concreta: implementa as operações primitivas para realizar passos do algoritmo que são específicos da subclasse.
- Colaborações: A Classe Concreta conta com a Classe Abstrata que implementa os passos invariante do algoritmo.
- Padrões Relacionados:
 - Método Fábrica: métodos-fábrica normalmente são chamados por métodos-modelo;
 - Estratégia: enquanto os métodos-modelo utilizam herança para variar partes de um algoritmo, as estratégias usam delegação para variar o algoritmo inteiro.

Procurador

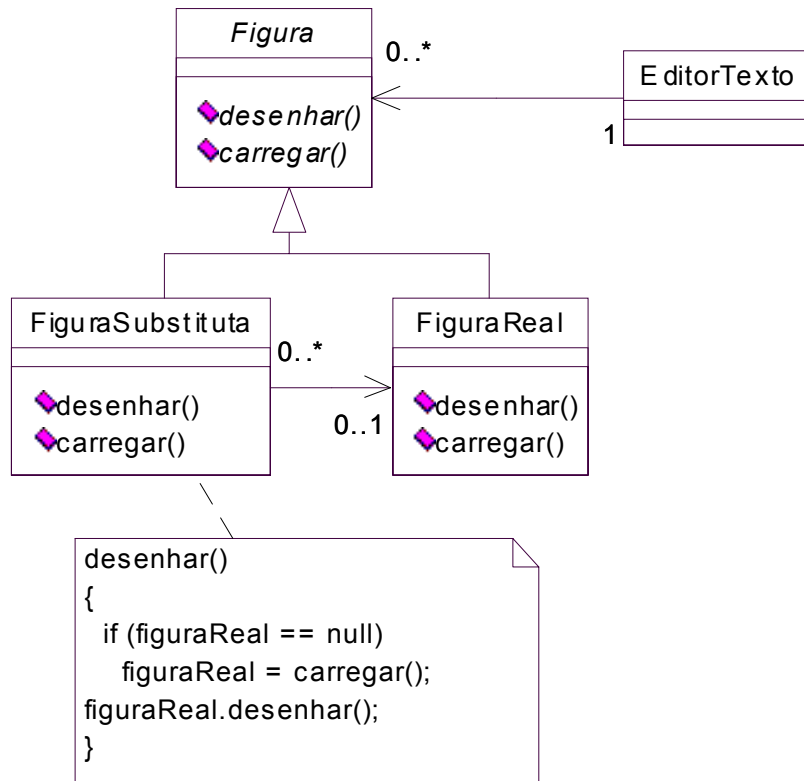
- Classificação: Padrão Estrutural de Objeto
- Propósito: provê um substituto/procurador que tem autorização para controlar o acesso ao objeto.
- Também conhecido como: Substituto, *Proxy*.
- Motivação: Uma razão para se controlar o acesso a um objeto é tentar adiar o alto custo de criação e inicialização deste objeto até o momento em que ele for ser realmente utilizado. Considere um editor de texto que pode embutir objetos gráficos em um documento. Alguns desses objetos, como figuras complexas, podem ter um alto custo de criação. Contudo, a abertura de um documento deve ser rápida. Assim, é desejável evitar a criação de todos esses objetos no momento em que o documento é aberto, até porque muitos deles não estarão visíveis ao mesmo tempo. Esta restrição sugere a criação dos objetos complexos sob demanda, isto é, o objeto só será criado no momento em que sua imagem se tornar visível. Mas o que colocar no documento no lugar da imagem? E como esconder essa abordagem sem complicar a implementação do editor? A solução consiste em criar um outro objeto, o procurador (*proxy*) da imagem, que atuará como substituto para a imagem real. Este procurador agirá exatamente como a imagem e cuidará de sua instanciação quando a mesma for requerida. O procurador da imagem criará a imagem real somente quando o editor de texto requisitar a ele que exiba a imagem, através da operação **desenhar()**. A partir daí, o procurador passará adiante as requisições subsequentes diretamente para a imagem, como ilustra o diagrama de seqüência abaixo.



- Aplicabilidade: O padrão Procurador é aplicável sempre que houver necessidade de uma referência mais versátil ou sofisticada do que um simples ponteiro. A seguir são listadas algumas situações nas quais este padrão é aplicável:
 - Um Procurador Remoto provê uma representação local para um objeto que se encontra em um outro espaço de endereçamento.
 - Um Procurador Virtual cria objetos complexos sob demanda, como no caso do exemplo da motivação.
 - Um Procurador de Proteção controla o acesso ao objeto original. Este tipo de procurador é amplamente utilizado quando há diferentes direitos de acesso ao objeto original. Neste caso, o procurador serve como uma espécie de filtro.
 - Um Procurador de Referência Inteligente é uma substituição para um ponteiro simples, que realiza operações adicionais quando o objeto é acessado. Isto pode ser útil em muitas situações, tais como:
 - contar o número de referências ao objeto real, de forma tal que ele possa ser liberado automaticamente quando não houver mais referências a ele;
 - carregar um objeto persistente para a memória quando ele for referenciado pela primeira vez;
 - verificar se o objeto real não está bloqueado antes de permitir um acesso a ele, garantindo que nenhum outro objeto poderá alterá-lo.
- Estrutura:



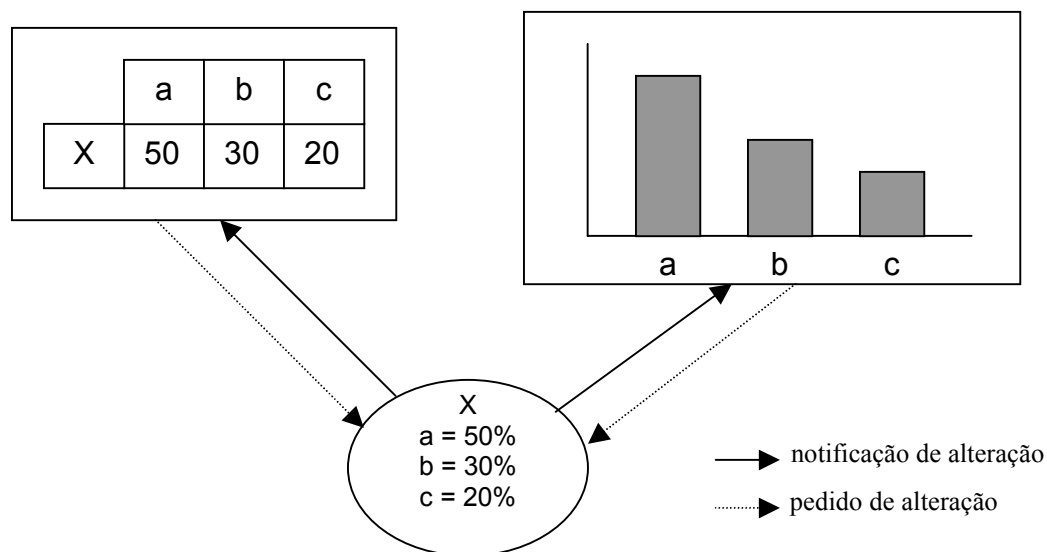
- Participantes:
 - Assunto: define uma interface comum para o ObjetoReal e para o Procurador, de modo que o procurador possa ser usado no lugar do objeto real;
 - Procurador: representa um substituto para o objeto real. Para tal, mantém uma referência ao objeto real, que o permite acessar este objeto. Sua interface deve ser idêntica à do objeto real, de modo que possa ser por ele substituído. Além disso, controla o acesso ao objeto real e pode ser responsável pela criação e exclusão do mesmo. Outras responsabilidades podem lhe ser atribuídas em função do tipo do procurador;
 - ObjetoReal: define o objeto real que o procurador representa.
- Colaborações: O procurador envia requisições para o objeto real, quando for apropriado, dependendo do tipo de Procurador.
- Conseqüências: O padrão Procurador introduz um nível de indireção quando se acessa o objeto. Esta indireção adicional tem muitos usos, dependendo do tipo de procurador. Um Procurador Remoto, por exemplo, pode esconder o fato de um objeto residir em um espaço de endereçamento diferente. Um Procurador Virtual pode realizar otimizações, como criar um objeto sob demanda. Procuradores de Proteção e de Referência Inteligente, por sua vez, permitem tarefas adicionais de gerenciamento quando um objeto é acessado.



- Padrões Relacionados:
 - Adaptador: um adaptador provê uma interface diferente para o objeto que ele adapta. O procurador provê a mesma interface.
 - Decorador: Um decorador adiciona responsabilidades a um objeto, enquanto o procurador controla o acesso ao objeto.

Observador

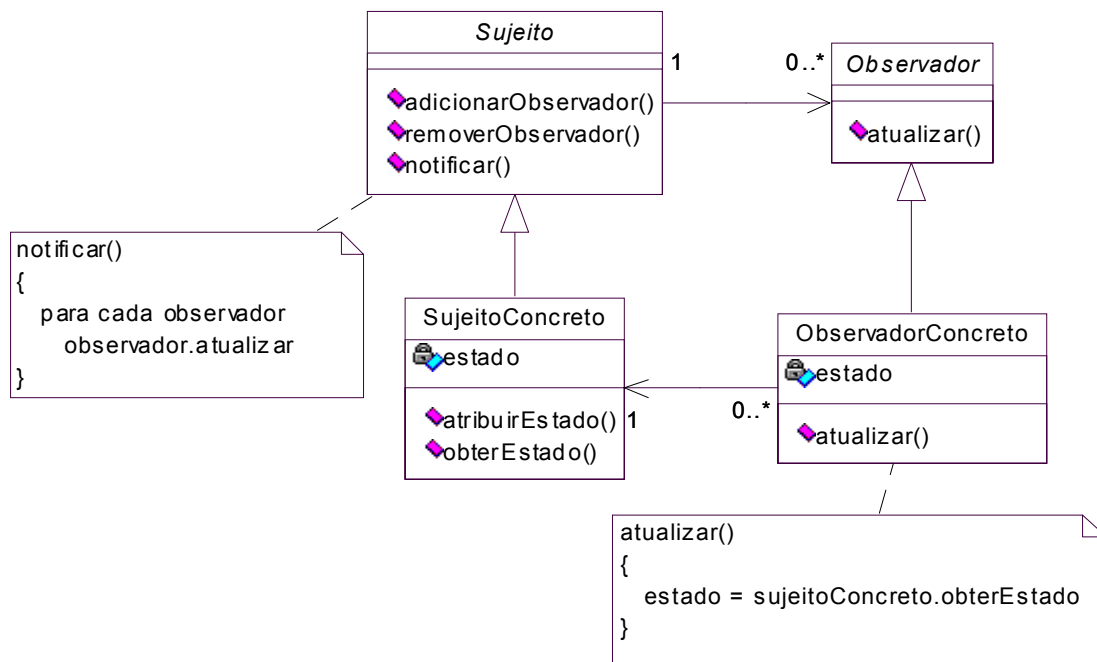
- Classificação: Padrão Comportamental de Objeto
- Propósito: define uma dependência um-para-muitos entre objetos de modo que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.
- Também conhecido como: Dependentes.
- Motivação: É uma boa opção para ferramentas de interfaces gráficas com o usuário separar aspectos de apresentação dos respectivos dados da aplicação. As classes dos componentes de domínio do problema e de interface com o usuário podem ser reutilizadas independentemente, assim como podem trabalhar juntas. Por exemplo, os mesmos dados estatísticos podem ser apresentados em formato de gráfico de barras ou planilha, usando apresentações diferentes. O gráfico de barras e a planilha devem ser independentes, de modo a permitir reuso. Contudo, eles têm de se comportar consistentemente, isto é, quando um usuário altera a informação na planilha, o gráfico de barras reflete a troca imediatamente e vice-versa.



Este comportamento implica que a planilha e o gráfico de barras são dependentes do mesmo objeto de dados e, portanto, devem ser notificados quando ocorre alguma mudança no estado desse objeto.

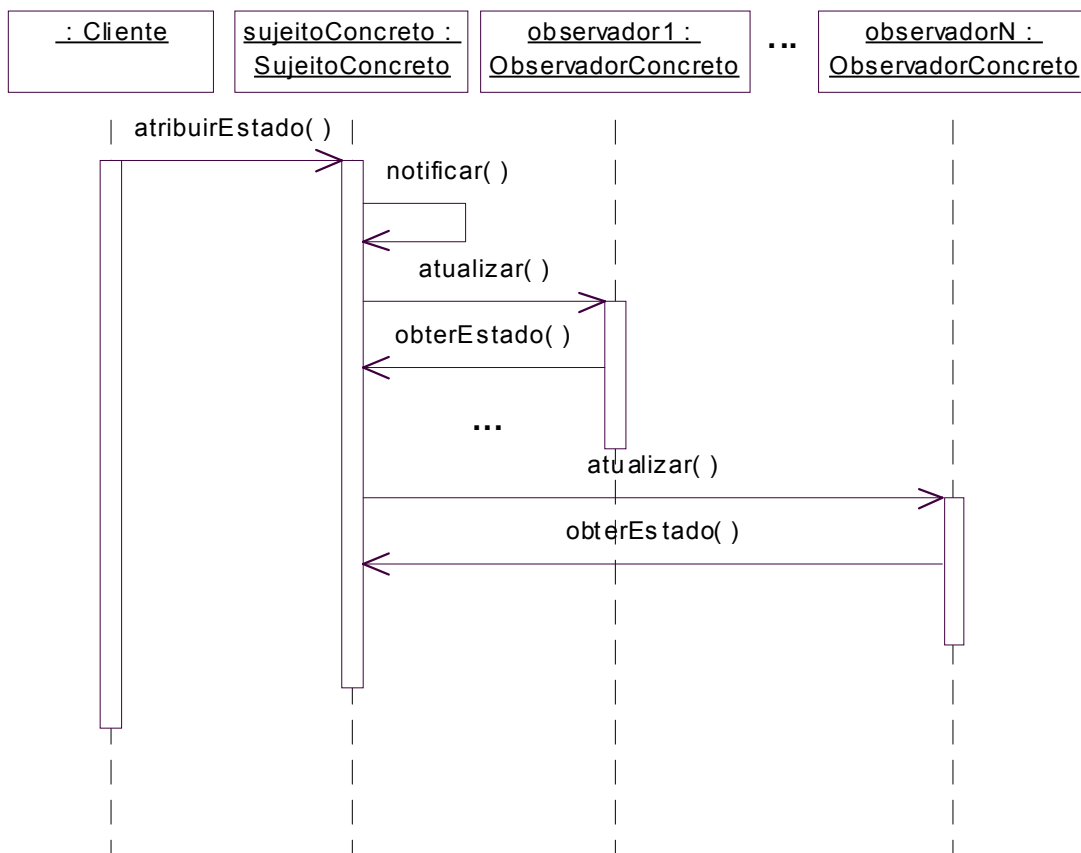
O padrão Observador descreve como se estabelecem estes relacionamentos. Os objetos principais deste padrão são Sujeito e Observador. O sujeito, no exemplo o objeto **X**, pode ter qualquer número de observadores, no caso a planilha e o gráfico de barras. Todos os observadores são notificados sempre que ocorre uma mudança no estado do sujeito. Em resposta, cada observador irá consultar o sujeito para sincronizar seu estado com o estado do sujeito.

- Aplicabilidade: O padrão Procurador é aplicável em qualquer uma das seguintes situações:
 - Quando uma abstração possui dois aspectos, um dependente do outro. Encapsular esses aspectos em objetos separados permite variá-los e reutilizá-los independentemente.
 - Quando uma alteração em um objeto requer alterações em outros e não se sabe quantos objetos precisam ser alterados.
 - Quando um objeto deveria ser capaz de notificar outros objetos sem fazer nenhuma suposição sobre como são esses objetos, ou seja, não se quer esses objetos fortemente acoplados.
- Estrutura:



- Participantes:
 - **Sujeito**: conhece seus observadores e provê uma interface para adicionar e remover objetos observadores. Qualquer número de observadores pode observar um sujeito.
 - **Observador**: define uma interface de atualização para os objetos que devem ser notificados das mudanças no sujeito.
 - **SujeitoConcreto**: armazena o estado de interesse para os observadores concretos e envia notificações para eles quando o seu estado é alterado.

- ObservadorConcreto: mantém uma referência para um objeto SujeitoConcreto, armazena o estado que deve ficar consistente com o estado do sujeito e implementa a interface de atualização do Observador, de modo a manter seu estado consistente com o do sujeito.
- Colaborações: O sujeito concreto notifica seus observadores sempre que ocorre uma alteração que pode tornar o estado de seus observadores inconsistente com o seu estado. Após ser informado de uma mudança no sujeito concreto, um objeto ObservadorConcreto pode consultar o sujeito, usando esta informação para reconciliar seu estado com o estado do sujeito.



- Consequências: O padrão Observador permite variar sujeitos e observadores independentemente. Deste modo é possível reutilizar sujeitos sem reutilizar observadores e vice-versa. Isso permite adicionar observadores sem modificar o sujeito ou outros observadores. Outros benefícios e obrigações desse padrão incluem:
 - Acoplamento abstrato entre Sujeito e Observador: Tudo que um sujeito sabe é que ele possui uma lista de observadores, todos em conformidade com a interface simples da classe abstrata Observador. O sujeito não conhece a classe concreta de nenhum observador. Assim, o acoplamento entre sujeitos e observadores é abstrato e mínimo.
 - Suporte para comunicação *broadcast*: Ao contrário de uma notificação individual, a notificação que um sujeito envia não precisa especificar seus receptores. A notificação é enviada automaticamente para todos os objetos interessados. Assim, a responsabilidade de um sujeito é limitada apenas à notificação de seus observadores. Isso oferece liberdade de adicionar e remover observadores a qualquer momento.
 - Atualizações inesperadas: Uma vez que os observadores não têm conhecimento da presença uns dos outros, eles podem não ser conscientes do custo de uma alteração no sujeito. Assim, uma operação aparentemente inócua sobre o sujeito pode provocar uma atualização em cascata para seus observadores e objetos dependentes. Além disso, critérios de dependência não bem definidos geralmente levam a atualizações falsas que podem ser difíceis de propagar.

Referências

- (Ambler, 1998) *Análise e Projeto Orientados a Objetos*. Scott Ambler, IBPI Press, 1998.
- (Coad et al., 1993) *Projeto Baseado em Objetos*, P. Coad, E. Yourdon, Editora Campus, 1993.
- (Gamma et al., 1995) *Design Patterns - Elements of Reusable Object-oriented Software*. Gamma, E., Helm R., Johnson R., Vlissides, J. Addison-Wesley Professional Computing Series, 1995.
- (Magela, 1998) *Produzindo Software Orientado a Objetos - Projeto*. Rogério Magela, Fuzion Engenharia de Software, 1998.
- (Pressman, 2002) *Engenharia de Software*. Roger S. Pressman, tradução da 5ª edição, Mc Graw Hill, 2002.
- (Yourdon, 1994) *Object-Oriented Systems Design: an Integrated Approach*, E. Yourdon, Yourdon Press Computing Series, Prentice Hall, 1994.

6. Projeto Estruturado de Sistemas

No projeto de sistemas segundo o paradigma estruturado, assim como no paradigma OO, um modelo de projeto é gerado a partir do modelo de análise, com o objetivo de representar o que deverá ser codificado na fase de implementação.

Uma vez que o paradigma estruturado trabalha com a clara distinção entre dados e funções, o projeto estruturado tende a seguir o mesmo caminho. Sendo assim, o projeto estruturado de sistema pode ser dividido em duas grandes fases: projeto de dados e projeto de programas.

6.1 - Projeto de Dados

Um aspecto fundamental da fase de projeto consiste em estabelecer de que forma serão armazenados os dados do sistema. Em função da plataforma de implementação, diferentes soluções de projeto devem ser adotadas. Isto é, se o software tiver de ser implementado em um banco de dados hierárquico, por exemplo, um modelo hierárquico deve ser produzido, adequando a modelagem de entidades e relacionamentos a esta plataforma de implementação.

Atualmente, a plataforma de implementação para armazenamento de dados mais difundida é a dos Bancos de Dados Relacionais e, portanto, neste texto, discutiremos apenas o projeto lógico de bancos de dados relacionais.

Conforme discutido no capítulo 4, em um modelo de dados relacional, os conjuntos de dados são representados por tabelas de valores. Cada tabela, denominada de relação, é bidimensional, sendo organizada em linhas e colunas.

Para se realizar o mapeamento de um modelo de entidades e relacionamentos em um modelo relacional, pode-se utilizar como ponto de partida as seguintes diretrizes:

- Conjuntos de entidades e agregados devem dar origem a tabelas;
- Uma instância de um conjunto de entidades ou de um agregado deve ser representada como uma linha da tabela correspondente;
- Um atributo de um conjunto de entidades ou agregado deve ser tratado como uma coluna da tabela correspondente;
- Toda tabela tem de ter uma chave primária, que pode ser um atributo determinante do conjunto de entidades ou agregado correspondente, ou uma nova coluna criada exclusivamente para este fim;
- Relacionamentos devem ser mapeados através da transposição da chave primária de uma tabela para a outra.

Ainda que este mapeamento seja amplamente aplicável, é sempre necessário avaliar requisitos não funcionais para se chegar ao melhor projeto para uma dada situação. Além disso, os relacionamentos requerem um cuidado maior e, por isso, são tratados a seguir com mais detalhes.

Relacionamentos 1 : 1

No exemplo da figura 6.1, ambas as soluções são igualmente válidas. Deve-se observar para cada caso, contudo, a melhor solução, considerando os seguintes aspectos:

- Se **A** for total em **R** (todo **A** está associado a um **B**), é melhor colocar a chave de **B** (**#B**) em **A**, como mostra o exemplo da figura 6.2.
- Se **B** for total em **R** (todo **B** está associado a um **A**), é melhor colocar a chave de **A** (**#A**) em **B**.
- Se ambos forem totais, pode-se trabalhar com uma única tabela escolhendo uma das chaves **#A** ou **#B** como chave primária, como mostra o exemplo da figura 6.3.
- Caso contrário, é melhor transpor a chave que dará origem a uma coluna mais densa, isto é, que terá menos valores nulos.

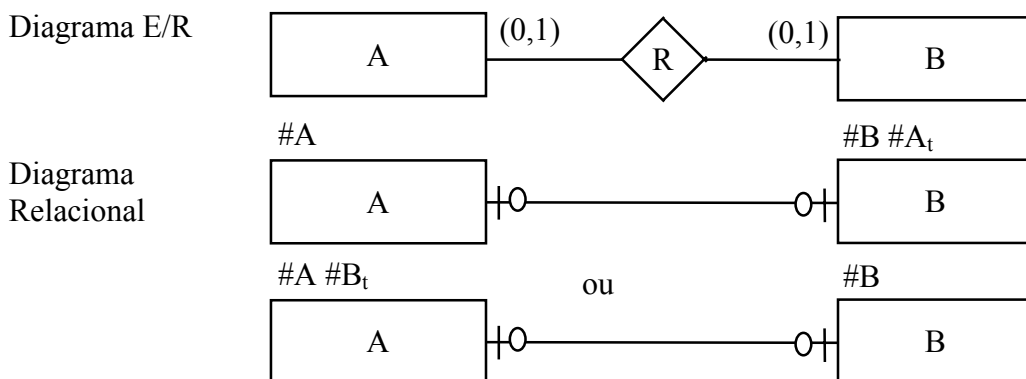


Figura 6.1 - Tradução de Relacionamentos 1:1 do Diagrama E/R para o Relacional.

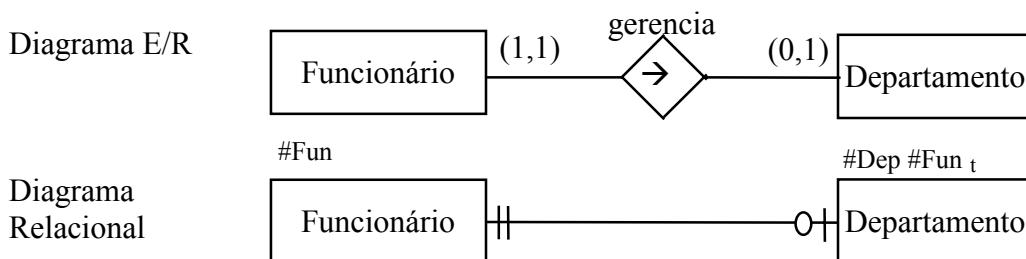
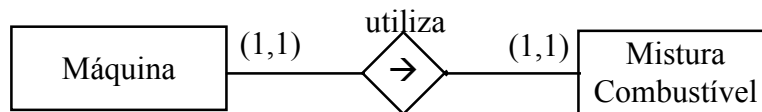


Figura 6.2 – Exemplo de Relacionamento 1:1.

Diagrama E/R



Uma máquina emprega necessariamente uma mistura combustível e vice-versa.

Diagrama Relacional

#Maq ou #MCo

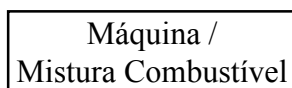


Figura 6.3 - Exemplo de um relacionamento 1:1 total em ambos os lados.

Relacionamentos 1 : N

Neste caso, deve-se transpor a chave da tabela correspondente ao conjunto de entidades de cardinalidade máxima N para a tabela que representa o conjunto de entidades cuja cardinalidade máxima é 1, como mostra a figura 6.4.

Diagrama E/R

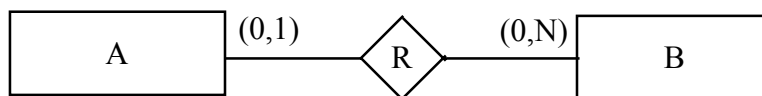


Diagrama Relacional



Figura 6.4 - Tradução de Relacionamentos 1:N do Diagrama E/R para o Relacional.

Um A pode estar associado a vários Bs, mas um B só pode estar associado a um A, logo deve-se transpor a chave primária de A para B. A figura 6.5 mostra um exemplo desta situação.

Diagrama E/R

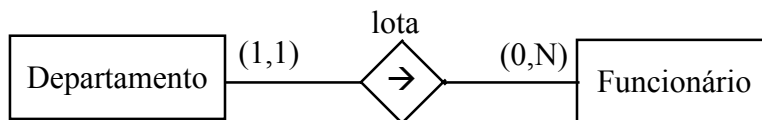


Diagrama Relacional

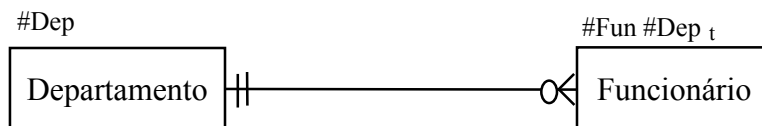


Figura 6.5 – Exemplo de um relacionamento 1:N.

Relacionamentos N : N

No caso de relacionamentos N:N (agregado), deve-se criar uma terceira tabela, transpondo as chaves primárias das duas tabelas que participam do relacionamento N:N, como mostra a figura 6.6. Se existirem atributos do relacionamento, estes deverão ser colocados na nova tabela. Caso seja necessário, algum desses atributos pode ser designado para compor a chave primária da tabela correspondendo ao agregado, como ilustra o exemplo da figura 6.7.

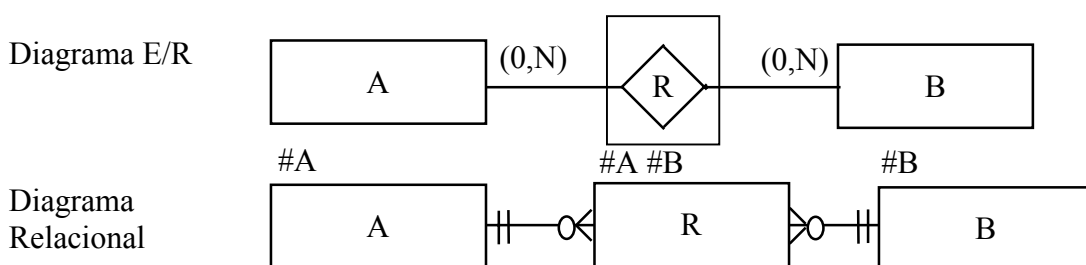


Figura 6.6 - Tradução de Relacionamentos N:N do Diagrama E/R para o Relacional.

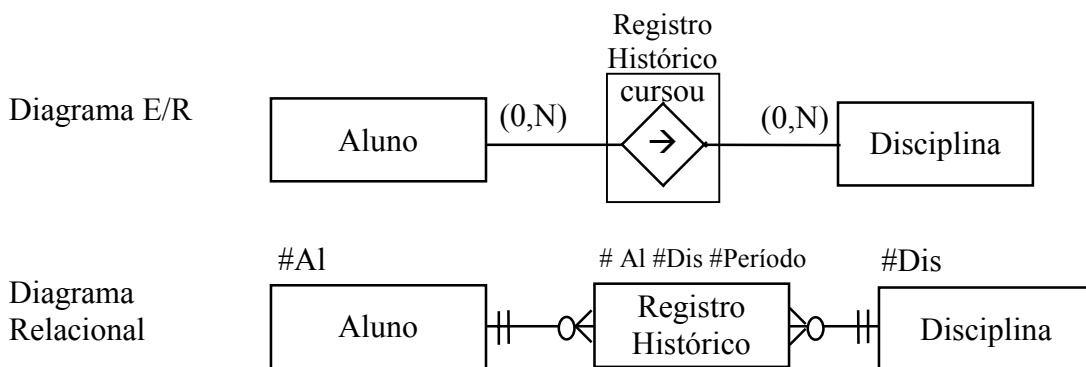


Figura 6.7 – Exemplo de relacionamento N:N.

Auto-Relacionamentos

Os auto-relacionamentos devem seguir as mesmas regras de tradução de relacionamentos, como mostram os exemplos das figuras 6.8 e 6.9.

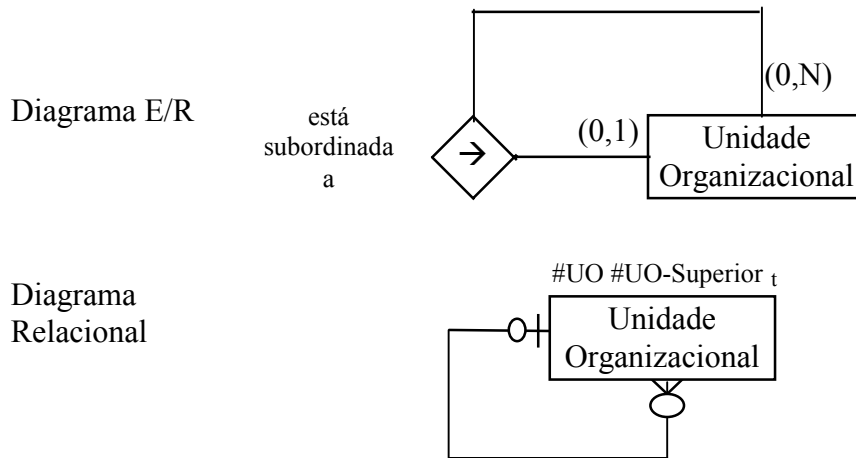


Figura 6.8 – Exemplo de auto-relacionamento 1:N.

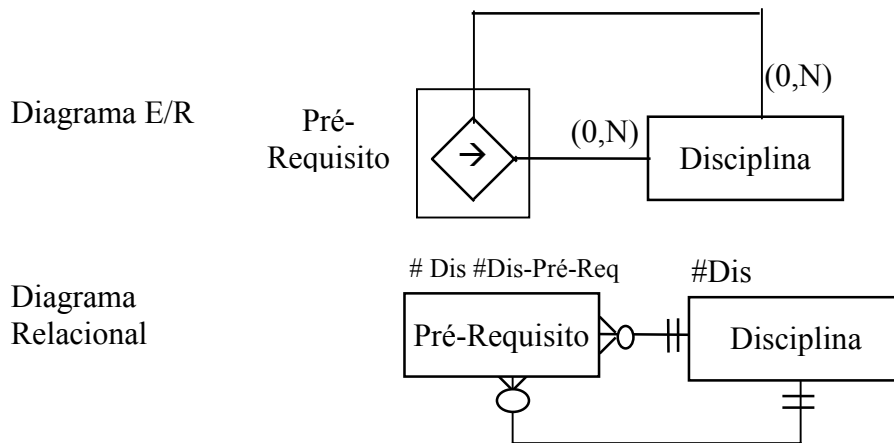
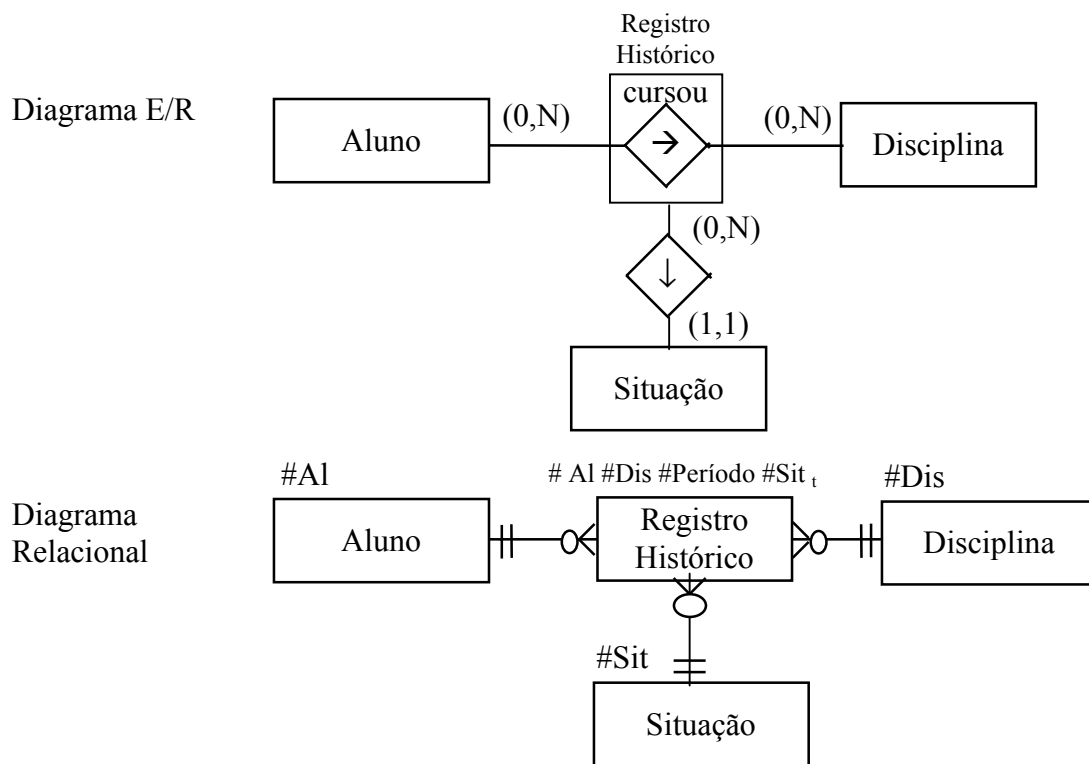


Figura 6.9 – Exemplo de auto-relacionamentos N:N.

Relacionamentos entre um Conjunto de Entidades e um Agregado

Já discutimos como fazer a tradução de um agregado para o modelo relacional. Um relacionamento entre uma entidade e um agregado terá o mesmo tratamento que um relacionamento entre entidades, considerando, agora, o agregado como uma entidade. Tomemos como exemplo um relacionamento 1:N entre uma entidade e um agregado, como mostra o exemplo da figura 6.10.



Relacionamento Ternário

No caso de relacionamentos ternários, deve-se criar uma nova tabela contendo as chaves das três entidades envolvidas, como mostra a figura 6.11. Assim como no caso de agregados, se existirem atributos do relacionamento, estes deverão ser colocados na nova tabela. Caso seja necessário, algum desses atributos pode ser designado para compor a chave primária da nova tabela.

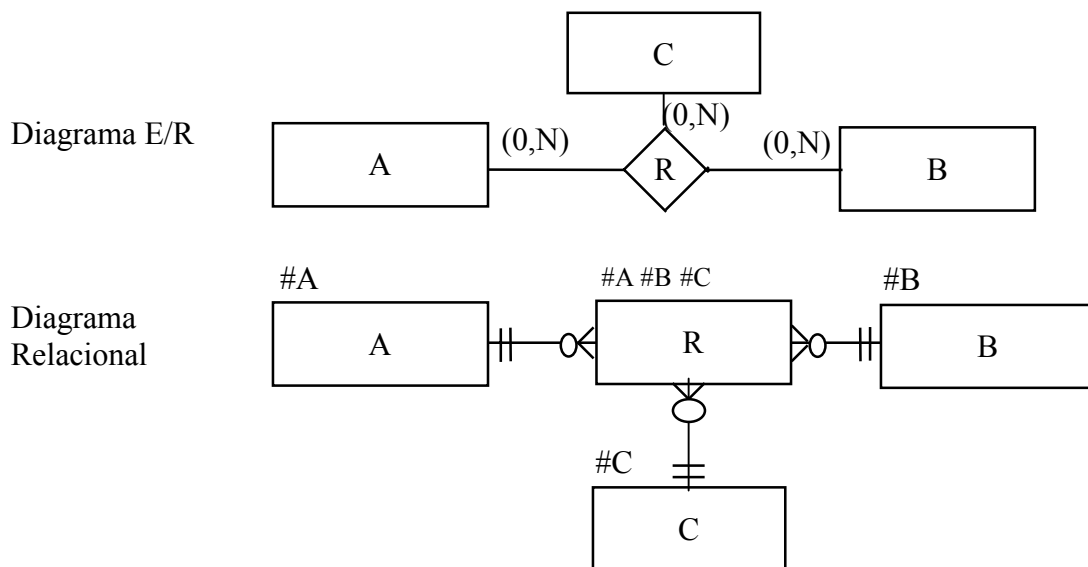


Figura 6.11 - Tradução de Relacionamentos Ternários.

Particionamento

No caso de particionamento de conjuntos de entidades, deve-se criar uma tabela para o super-tipo e tantas tabelas quantos forem os sub-tipos, todos com a mesma chave, como mostra a figura 6.12. Caso não haja no modelo conceitual um atributo determinante no super-tipo, uma chave primária deve ser criada para fazer a amarração com os sub-tipos.

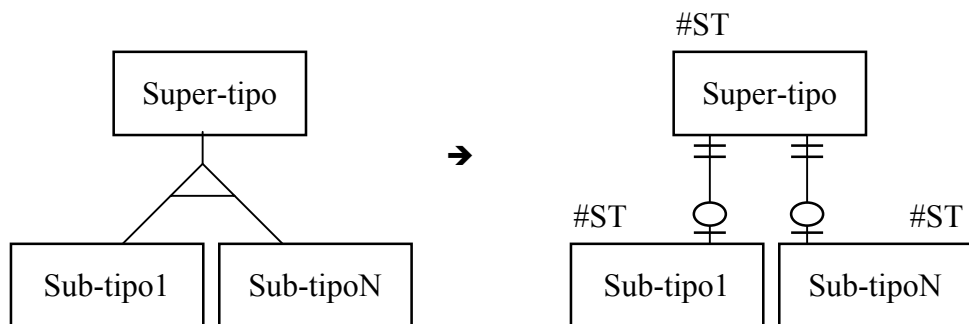


Figura 6.12 – Tradução de Particionamento.

Atributos Multivalorados

Segundo a propriedade do modelo relacional que nos diz que cada célula de uma relação pode conter no máximo um único valor, não podemos representar atributos multivalorados como uma única coluna da tabela. Há algumas soluções possíveis para este problema, tal como, criar tantas colunas quantas necessárias para representar o atributo. Esta solução, contudo, pode, em muitos casos, não ser eficiente ou mesmo possível. Uma solução mais geral para este problema é criar uma tabela em separado, como mostra a figura 6.13.

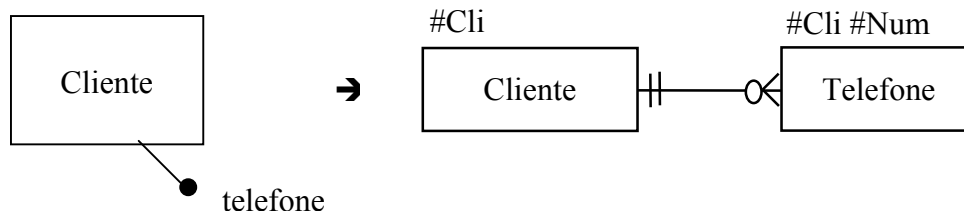


Figura 6.13 – Mapeamento Geral de Atributos Multi-valorados.

6.2 - Projeto Modular de Programas

A tarefa de construção de sistemas computadorizados requer uma organização das idéias, de modo a se conseguir desenvolver produtos com qualidade. Programas escritos sem qualquer subdivisão são inviáveis do ponto de vista administrativo e não permitem reaproveitamento de trabalhos anteriormente executados.

O Projeto Modular de Programas oferece uma coleção de orientações, técnicas, estratégias e heurísticas capazes de conduzir a bons projetos de programas. O objetivo é desenvolver programas com menor complexidade, usando o princípio “dividir para simplificar”. Como resultados de um bom projeto de programas, tem-se:

- Facilidade na leitura de programas (maior legibilidade);
- Maior rapidez na depuração de programas na fase de testes;
- Facilidade de modificação de programas na fase de manutenção (maior alterabilidade).

O projeto estruturado de sistemas, em sua dimensão de funções, envolve duas grandes etapas: o projeto da arquitetura do sistema e o projeto detalhado dos módulos. Paralelamente, devem ser feitos o projeto de dados e o projeto da interface com o usuário.

Em ambos os casos do projeto funcional estruturado, técnicas de Projeto Modular de Programas são empregadas. Apenas de se usar diferentes variações, basicamente, dois conceitos são centrais para o projeto estruturado de sistemas:

- **Módulo:** Conjunto de instruções que desempenha uma função específica dentro de um programa. É definido por: entrada/saída, função, lógica e dados internos.
- **Conexão entre Módulos:** Forma como os módulos interagem entre si.

O bloco básico de construção de um programa estruturado é um módulo. Os programas estruturados são organizados como uma hierarquia de módulos. A idéia básica, então, é estruturar os programas em termos de módulos e conexões entre estes módulos.

O Projeto Modular de Programas considera, ainda, cinco aspectos importantes para o projeto de programas:

- Permite que a forma da solução seja guiada pela forma do problema.
- Procura solucionar sistemas complexos através da segmentação deste sistema em “caixas pretas” e pela organização destas “caixas pretas” em uma hierarquia conveniente para uma implementação computadorizada.
- Utiliza ferramentas, especialmente as gráficas, que tornam os sistemas de fácil compreensão.
- Oferece um conjunto de estratégias para desenvolver o projeto de solução a partir de uma declaração bem definida do problema.
- Oferece um conjunto de critérios para avaliação da qualidade de um determinado projeto-solução com respeito ao problema a ser resolvido.

São objetivos do Projeto Modular de Programas:

- Permitir a construção de programas mais simples;
- Obter módulos independentes;
- Permitir testes por partes;
- Ter menos código a analisar em uma manutenção;
- Servir de guia para a programação estruturada;
- Construir módulos com uma única função;
- Permitir reutilização.

6.2.1 - Simplificando um Sistema

O Projeto Modular procura simplificar um sistema complexo, particionando-o em módulos e organizando esses hierarquicamente. O sistema é subdividido em caixas-pretas, que são organizadas em uma hierarquia conveniente. A vantagem do uso da caixa-preta está no fato de que não precisamos conhecer como ela trabalha, mas apenas utilizá-la. As características de uma caixa-preta são:

- sabemos como devem ser os elementos de entrada, isto é, as informações necessárias para seu processamento;
- sabemos como devem ser os elementos de saída, isto é, os resultados oriundos do seu processamento;
- conhecemos a sua função, isto é, que processamento ela faz sobre os dados de entrada para que sejam produzidos os resultados;
- não precisamos conhecer como ela realiza as operações, nem tão pouco seus procedimentos internos, para podermos utilizá-la.

Sistemas compostos por caixas pretas são facilmente construídos, testados, corrigidos, entendidos e modificados. Deste modo, o primeiro passo no controle da

complexidade no projeto estruturado consiste em segmentar um sistema em caixas pretas de modo a atingir as seguintes metas:

- cada caixa preta deve resolver uma parte bem definida do problema;
- a função de cada caixa preta deve ser facilmente compreendida;
- conexões entre caixas pretas devem refletir apenas conexões entre partes do problema;
- as conexões devem ser tão simples e independentes quanto possível.

Organizando as Caixas Pretas Hierarquicamente

Antes de iniciarmos uma discussão sobre Projeto Modular de Programas, passemos a observar os exemplos das figuras 6.14, 6.15 e 6.16, que mostram três organogramas de empresas.

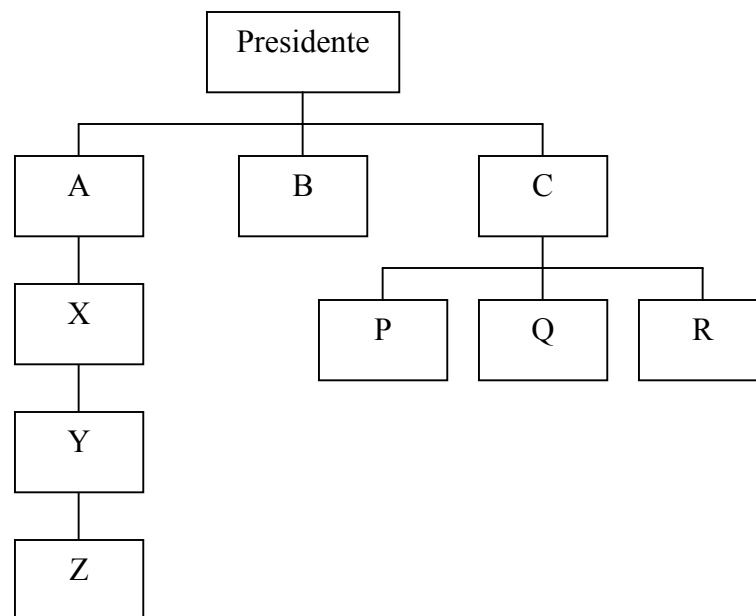


Figura 6.14 - Organograma da Empresa 1.

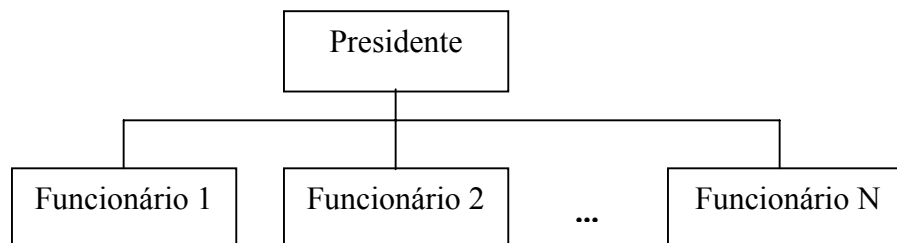


Figura 6.15 - Organograma da Empresa 2.

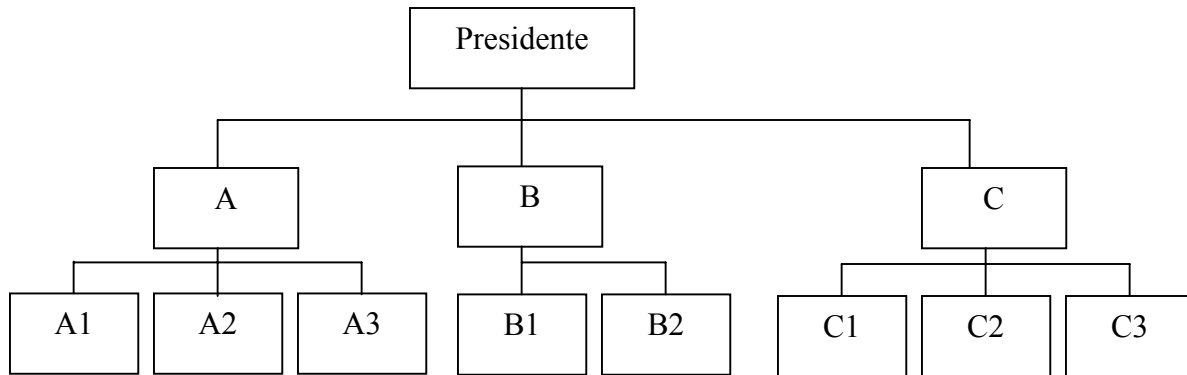


Figura 6.16 - Organograma da Empresa 3.

Como podemos notar, no organograma da empresa 1, o vice-presidente *A* e os gerentes *X* e *Y*, possuem tarefas triviais, pois cada um deles tem como responsabilidade gerenciar apenas um subordinado. Neste caso, todo serviço seria realizado pelo funcionário *Z*. Poderíamos sugerir, então, acabar com as gerências. Por outro lado, o presidente da empresa 2 está sobrecarregado, uma vez que ele gerencia funcionários demais. A empresa 3 parece apresentar um organograma mais equilibrado, no qual cada gerente gerencia um número apropriado de subordinados.

As estruturas de um programa, ou de um sistema, podem ser discutidas de maneira análoga à questão dos organogramas. Ou seja, os módulos (caixas-pretas) devem ser dispostos em uma hierarquia de modo a, por um lado, não provocar sobrecarga de processamento e, de outro, não criar módulos apenas intermediários, sem desempenhar nenhuma função.

Há vários tipos de diagramas hierárquicos para o projeto de programas (Martin et al., 1991). Neste texto, serão explorados dois deles: o Diagrama Hierárquico de Funções (DHF), usado principalmente para o projeto arquitetural, e o Diagrama de Estrutura Modular (DEM), usado para o projeto detalhado de módulos. A diferença básica entre eles é que o DHF não representa o fluxo de dados e controles entre módulos, nem aspectos relacionados com detalhes lógicos de um módulo, tais como estruturas de repetição (laços) e condição. Estas informações são capturadas em um DEM e, por isso mesmo, o DEM é empregado no projeto detalhado de módulos, enquanto o DHF é usado para o projeto da arquitetura do sistema.

6.2.2 - Diagrama Hierárquico de Funções

Um Diagrama Hierárquico de Funções (DHF) define a arquitetura global de um programa ou sistema, mostrando módulos e suas inter-relações (Martin et al., 1991). Cada módulo pode representar um subsistema, programa ou módulo de programa. Sua finalidade é mostrar os componentes funcionais gerais (arquitetura do sistema) e fazer referência a diagramas detalhados (tipicamente Diagramas de Estrutura Modular). Um DHF não mostra o fluxo de dados entre componentes funcionais ou qualquer informação de estruturas de controle, tais como laços (*loops*) ou condições.

A estrutura de um DHF tem como ponto de partida um módulo inicial, localizado no topo da hierarquia, que detém o controle sobre os demais módulos do diagrama, ditos seus módulos-filhos. Um módulo-filho, por sua vez, pode ser “pai” de outros módulos, indicando que ele detém o controle sobre esses módulos.

A construção de um DHF pode ser bastante facilitada se existir um diagrama de casos de uso do sistema. De fato, considerações análogas às feitas no projeto da Componente de Gerência de Tarefas podem ser aplicadas no projeto arquitetural usando DHFs. Cada executável deve dar origem a um DHF. Os casos de uso controlados por esse executável devem ser módulos-filhos do módulo inicial do diagrama. Cenários de um caso de uso podem ser representados como módulos-filhos do módulo correspondente. Para sistemas de médio a grande porte, contudo, representar todos os casos de uso e seus cenários em um único diagrama pode torná-lo muito complexo. Assim, novos DHFs poderiam ser elaborados para cada caso de uso, ou para alguns casos de uso.

Tomemos como exemplo um sistema de entrega à domicílio de refeições, cujo diagrama de casos de uso é apresentado na figura 6.17.

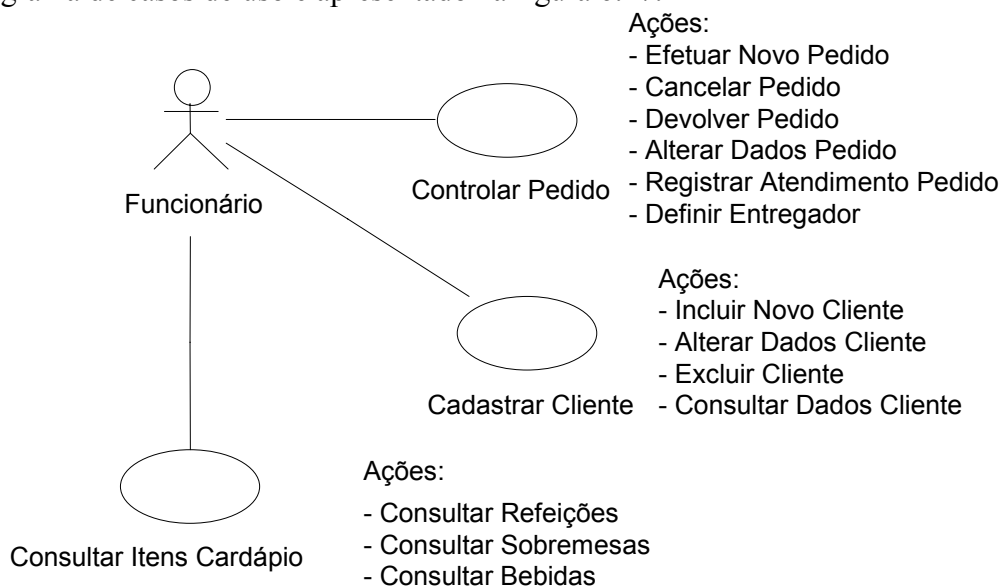


Figura 6.17 – Diagrama de Casos de Uso de um Sistema de Entrega de Refeições à Domicílio.

Com base neste modelo, poderíamos construir o DHF mostrado na figura 6.18. Neste diagrama, optou-se por não representar os cenários do caso de uso *Controlar Pedido*, uma vez que este é um caso de uso bastante complexo, com vários cenários, o que traria uma complexidade indesejada para o DHF. Assim, além do diagrama da figura 6.18, um outro, cujo módulo inicial seria *Controlar Pedido*, deveria ser elaborado.

Vale ressaltar que, assim como a Componente de Gerência de Tarefas, no projeto orientado a objetos, tem forte relação com a Componente de Interface com o Usuário, um DHF pode ser usado como um guia para o projeto das interfaces com o usuário, apoiando a definição de janelas, estruturas de menu, etc.

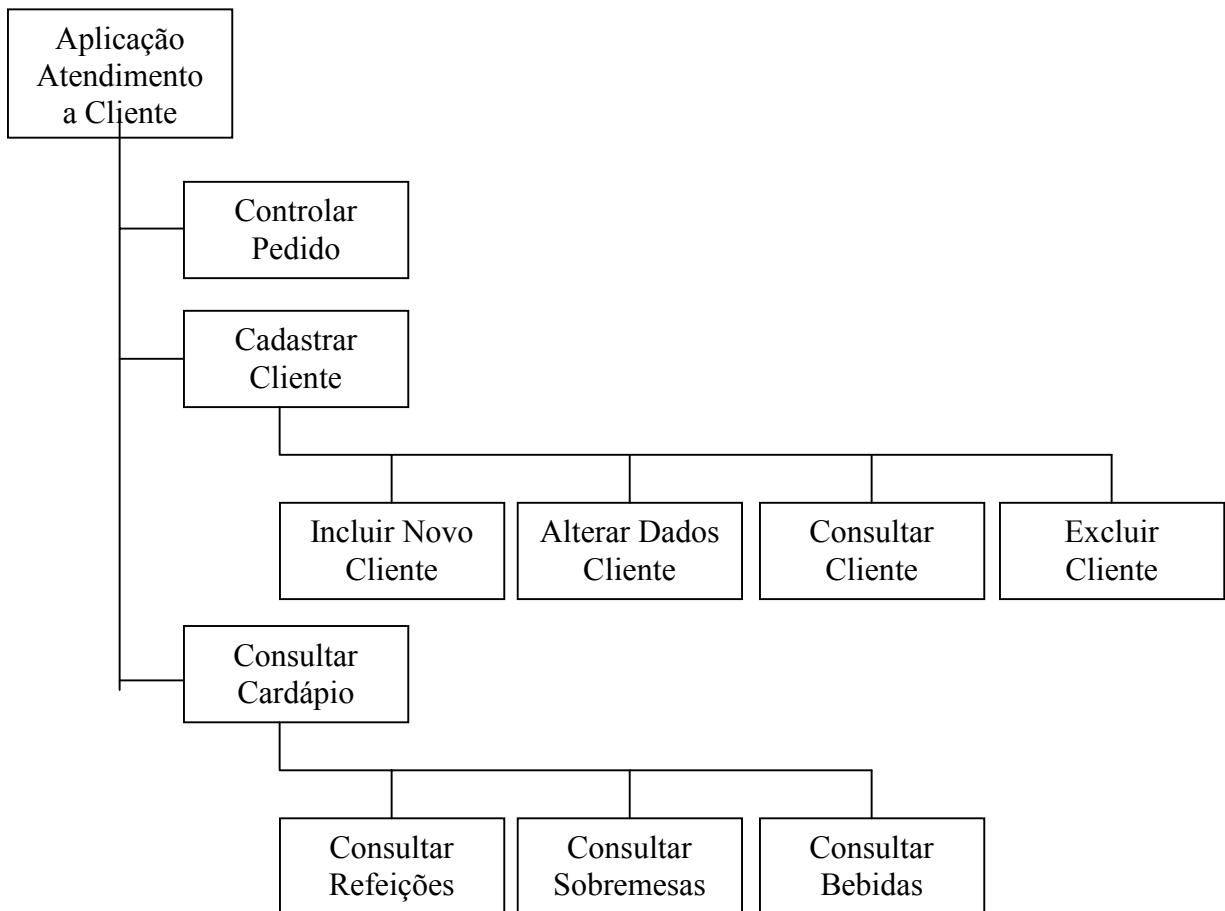


Figura 6.18 – Um Diagrama Hierárquico de Funções para o Sistema de Entrega de Refeições.

6.2.3 - Diagrama de Estrutura Modular

Em um Diagrama de Estrutura Modular (DEM), um programa é representado como um conjunto de módulos organizados hierarquicamente, de modo que os módulos que executam tarefas de alto nível no programa são colocados nos níveis superiores da hierarquia, enquanto os módulos que executam tarefas detalhadas, de nível mais baixo, aparecem nos níveis inferiores. Observando a hierarquia, os módulos a cada nível sucessivo contêm tarefas que definem as tarefas realizadas no nível precedente (Martin et al., 1991).

Um módulo é definido como uma coleção de instruções de programa com quatro atributos básicos: entradas e saídas, função, lógica e dados internos. Entradas e saídas são, respectivamente, as informações que um módulo necessita e fornece. A função de um módulo é o que ele faz para produzir, a partir da informação de entrada, os resultados da saída. Entradas, saídas e função fornecem a visão externa do módulo e, portanto, apenas estes aspectos são representados no diagrama de estrutura modular.

A lógica de um módulo é a descrição dos algoritmos que executam a função. Dados internos são aqueles referenciados apenas dentro do módulo. Lógica e dados internos representam a visão interna do módulo e são descritos por uma técnica de especificação de programas, tal como português estruturado, pseudocódigo, tabelas de decisão e árvores de decisão.

Assim sendo, um DEM mostra:

- O particionamento de um programa em módulos;
- A hierarquia e a organização dos módulos;
- As interfaces de comunicação entre módulos (entrada/saída);
- As funções dos módulos, dadas por seus nomes;
- Estruturas de controle entre módulos, tais como condição de execução de um módulo, laços de repetição de módulos (iteração), dentre outras.

Um DEM não mostra a lógica e os dados internos dos módulos e, por isso, deve ser acompanhado de uma descrição dos módulos, mostrando os detalhes internos dos procedimentos das caixas pretas.

Simbologia do DEM

A seguir, são apresentadas as principais notações utilizadas para elaborar Diagramas de Estrutura Modular (Xavier et al., 1995):

- **Módulo:** Em um DEM, um módulo é representado por um retângulo, dentro do qual está contido seu nome, como mostra a figura 6.19. Um módulo pré-definido é aquele que já existe em uma biblioteca de módulos e, portanto, não precisa ser descrito ou detalhado.

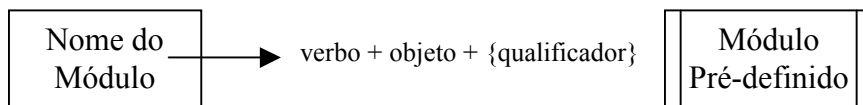


Figura 6.19 – Simbologia para Módulos em um DEM.

- **Conexão entre módulos:** Um sistema é um conjunto de módulos organizados dentro de uma hierarquia, cooperando e se comunicando para realizar um trabalho. A hierarquia mostra “quem chama quem”. Portanto, módulos devem estar conectados. No exemplo da figura 6.20, o módulo *A* chama o módulo *B* passando, como parâmetros, os dados *X* e *Y*. O módulo *B* executa, então, sua função e retorna o controle para *A*, no ponto imediatamente após à chamada de *B*, passando como resultado o dado *Z*. A ordem de chamada é sempre de cima para baixo, da esquerda para a direita.

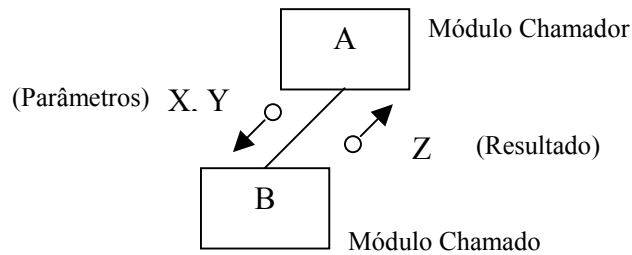


Figura 6.20 – Conexão entre módulos.

- **Comunicação entre módulos:** Módulos conectados estão se comunicando, logo existem informações trafegando entre eles. Estas informações podem ser dados ou controles (descrevem uma situação ocorrida durante a execução do módulo). A figura 6.21 mostra a convenção utilizada para se determinar se a informação que está sendo passada entre módulos é um dado ou um controle, juntamente com um exemplo.

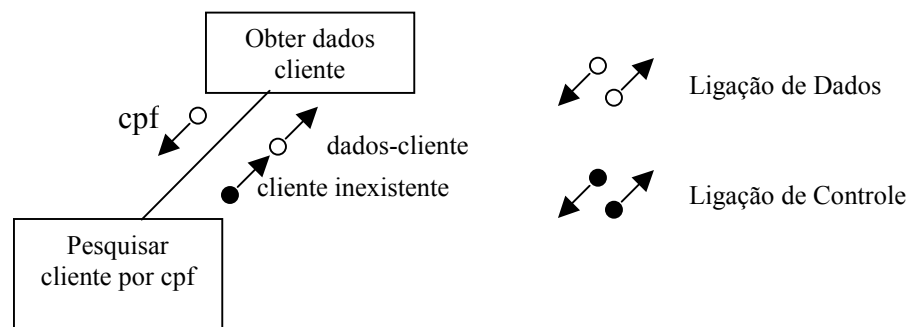


Figura 6.21 – Comunicação entre módulos.

- **Chamadas Condicionais:** Em muitos casos, um módulo só será ativado se uma condição for satisfeita. Nestes casos, temos chamadas condicionais, cuja notação é mostrada na figura 6.22. No exemplo à esquerda, o módulo A pode ou não chamar o módulo B. No exemplo à direita, o módulo A pode chamar um dos módulos B ou C.

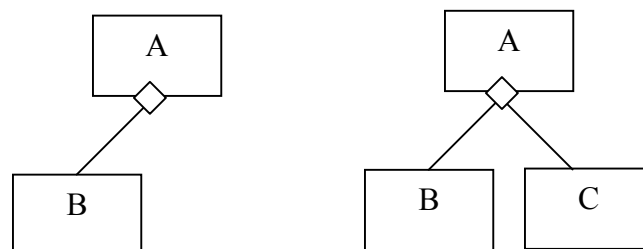


Figura 6.22 – Chamada Condicional.

- **Chamadas Iterativas:** Algumas vezes, nos deparamos com situações nas quais um módulo (ou um conjunto de módulos) é chamado várias vezes, caracterizando chamadas iterativas ou repetidas, cuja notação é mostrada na figura 6.23. No exemplo, os módulos *B* e *C* são chamados repetidas vezes pelo módulo *A*.

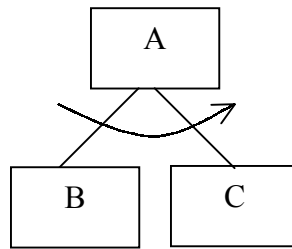


Figura 6.23 – Chamada Iterativa.

- **Conectores:** Algumas vezes, um mesmo módulo é chamado por mais de um módulo, às vezes em diagramas diferentes. Outras, o diagrama está complexo demais e deseja-se continuá-lo em outra página. Nestas situações, conectores podem ser utilizados, como ilustra a figura 6.24.

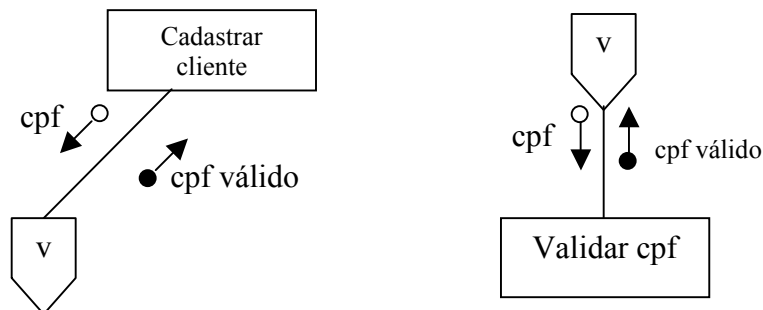


Figura 6.24 – Conectores.

Técnicas de Desenho

Para elaborar um diagrama de estrutura modular, devemos observar as seguintes orientações:

- Os módulos devem ser desenhados na ordem de execução, da esquerda para a direita.
- Cada módulo só deve aparecer uma única vez no diagrama. Para se evitar cruzamento de linhas, deve-se usar conectores.
- Não segmentar demais.

Além dessas orientações, o projeto estruturado fornece duas estratégias de projeto para guiar a elaboração de DEMs: a *análise de transformação* e a *análise de transação*.

Essas duas estratégias fornecem dois modelos de estrutura que podem ser usados isoladamente ou em combinação para derivar um projeto estruturado (Martin et al., 1991).

A **análise de transformação** é um modelo de fluxo de informações centrado na filosofia entrada-processamento-saída. Assim, o DEM correspondente tende a espelhar esta mesma estrutura, podendo ser decomposto em três grandes ramos, como mostra a figura 6.25.

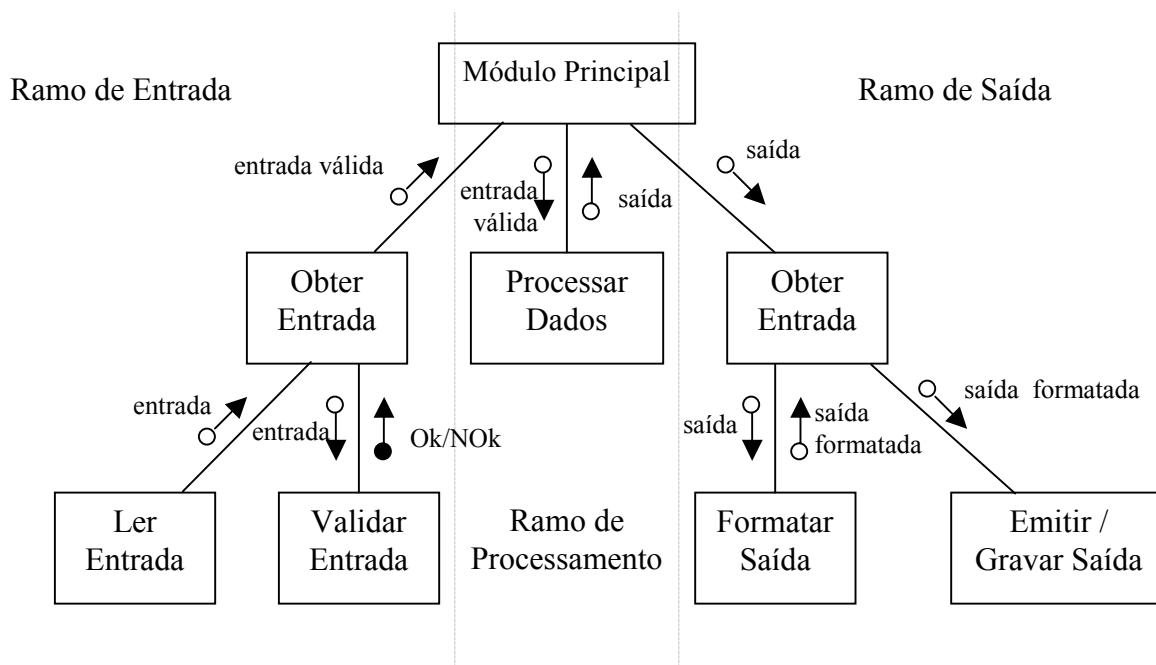


Figura 6.25 – Análise de Transformação.

O ramo de entrada contém os módulos que tratam da leitura e validação dos dados de entrada, bem como de uma eventual transformação para um formato adequado para o processamento. O ramo de processamento contém o processamento essencial e deve ser independente de considerações físicas de entrada e saída. Finalmente, o ramo de saída trata da transformação dos dados de saída de um formato interno para um formato adequado para o seu registro (p.ex., uma interface com o usuário ou um registro em bancos de dados).

A **análise de transação** é uma estratégia de projeto alternativa para a análise de transformação. Ela é útil no projeto de programas de processamento de transações. O DEM geral para a análise de transação é mostrado na figura 6.26. No topo do diagrama está um módulo centro de transação, que é responsável pela determinação do tipo de transação e pela chamada do módulo de transação apropriado. Abaixo dele, estão os vários módulos de transação. Há um módulo de transação para cada tipo de transação (Martin et al., 1991).

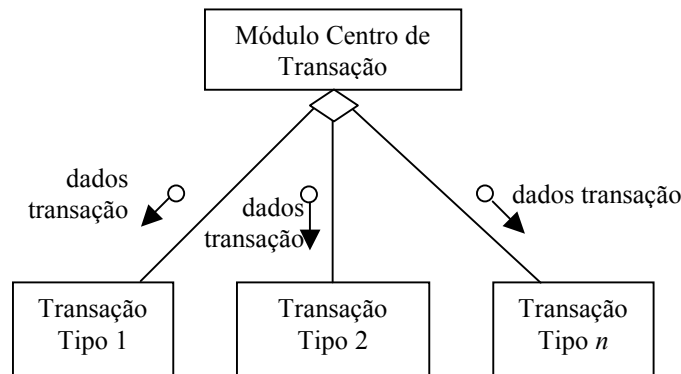


Figura 6.26 – Análise de Transação.

6.2.4 - Critérios de Qualidade de Projetos de Programa

O objetivo maior do projeto modular de programas é permitir que um sistema complexo seja particionado em módulos simples. No entanto, é vital que essa partição seja feita de tal forma que os módulos sejam tão independentes quanto possível e que cada um deles execute uma única função. Critérios que tratam estes aspectos são, respectivamente, acoplamento e coesão.

Acoplamento diz respeito ao grau de interdependência entre dois módulos. O objetivo é minimizar o acoplamento, isto é, tornar os módulos tão independentes quanto possível. Um baixo acoplamento pode ser obtido:

- eliminando relações desnecessárias;
- enfraquecendo a dependência das relações necessárias.

Podemos citar como razões para se minimizar o acoplamento:

- Quanto menos conexões houver entre dois módulos, menor será a chance de um problema ocorrido em um deles se refletir em outros.
- Uma alteração do usuário deve afetar o menor número de módulos possível, isto é, uma alteração em um módulo não deve implicar em alterações em outros módulos.
- Ao dar manutenção em um módulo, não devemos nos preocupar com detalhes de codificação de outros módulos.

O acoplamento envolve três aspectos principais: tipo da conexão, tamanho da conexão e o que é comunicado através da conexão. O tipo da conexão diz respeito à forma como uma conexão é estabelecida. O ideal é que a comunicação se dê através de chamadas a subrotinas, cada uma delas fazendo uso apenas de variáveis locais. Qualquer informação externa necessária deve ser passada como parâmetro. Assim, cada módulo deve possuir seu escopo próprio de variáveis, evitando-se utilizar uma variável definida em outro módulo.

Com relação ao tamanho da conexão, quanto menor o número de informações trafegando de um módulo para outro, menor será o acoplamento. Entretanto, vale a pena ressaltar que é importante manter-se a clareza da conexão. Não devemos mascarar as informações que fluem.

Finalmente, no que tange ao que é comunicado entre módulos, o ideal é que se busque acoplamento apenas de dados. Entretanto, quando se fizer necessária a comunicação de controles, devemos fazê-la sem máscaras. Seja o exemplo da figura 6.27.

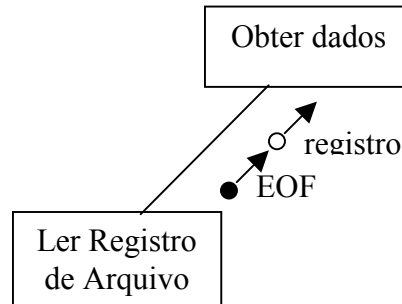


Figura 6.27 – Clareza na comunicação.

Neste caso, não é indicado mover brancos para o registro e se o registro estiver em branco é porque acabou o arquivo (EOF). Com este artifício, estar-se-ia mascarando o controle.

De maneira geral, não adianta melhorar dois destes aspectos se estivermos piorando o terceiro. Muitas vezes, o acoplamento resultante poderá ser maior. Só devemos fazer alterações que melhorem um dos aspectos sem afetar os demais. As seguintes orientações podem servir para apoiar a tomada de decisão:

- O módulo chamador não deve nunca enviar um controle ao módulo chamado. Isto significa que o módulo chamador está dizendo o que o módulo chamado deve fazer, caracterizando, portanto, que o módulo chamado não trata de uma única função.
- Só utilizar controles de baixo para cima. O módulo chamado avisa que não conseguiu executar sua função, mas não deve dizer ao chamador o que fazer.
- Evitar o uso de dados globais. Sempre que possível, utilizar variáveis locais.
- É inadmissível que um módulo se refira a uma parte interna de outro.

Em suma, para minimizar o acoplamento, devemos:

- Passar o menor número possível de parâmetros e, de preferência, apenas dados. Quando for necessário passar controles, fazê-lo apenas de baixo para cima.
- Ter pontos únicos de entrada e saída em um módulo.
- Sempre que possível, utilizar programas compilados separadamente.

Coesão define como as atividades de um módulo estão relacionadas umas com as outras. Vale a pena ressaltar que coesão e acoplamento são interdependentes e, portanto, uma boa coesão deve nos levar a um pequeno acoplamento. A figura 6.28 procura mostrar este fato.

No projeto modular de programas, os módulos devem ter alta coesão, isto é, seus elementos internos devem estar fortemente relacionados uns com os outros.

O grau de coesão de um módulo tem um impacto direto na qualidade do software produzido, sobretudo no que tange à alterabilidade, manutenibilidade, legibilidade e capacidade de reutilização. O ideal é que tenhamos apenas coesão funcional, isto é, que todos os elementos de um módulo estejam contribuindo para a execução de uma e somente uma função do sistema.

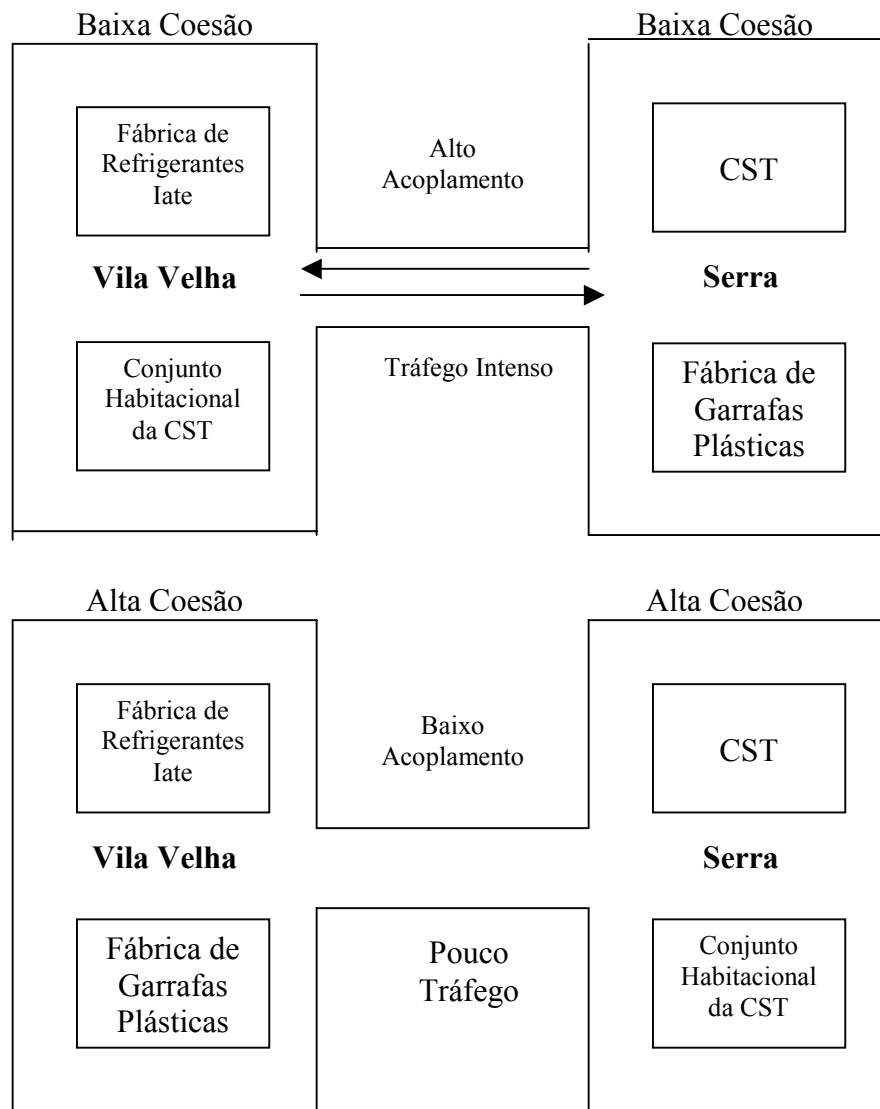


Figura 6.28 – Coesão e Acomplamento.

Referências

- (Martin et al., 1991) J. Martin, C. McClure. *Técnicas Estruturadas e CASE*. Makron Books, São Paulo, 1991.
- (Xavier et al., 1995) C.M.S. Xavier, C. Portilho. *Projetando com Qualidade a Tecnologia de Sistemas de Informação*. Livros Técnicos e Científicos Editora, 1995.