UFES - Universidade Federal do Espírito Santo

Engenharia de Software

Notas de Aula

Ricardo de Almeida Falbo

E-mail: falbo@inf.ufes.br

2005

1 – Introdução

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

Para os iniciantes na Ciência de Computação, desenvolver software é, muitas vezes, confundido com programação. Essa confusão inicial pode ser atribuída, parcialmente, pela forma como as pessoas são introduzidas nesta área de conhecimento, começando por desenvolver habilidades de raciocínio lógico, através de programação e estruturas de dados. Aliás, nada há de errado nessa estratégia. Começamos resolvendo pequenos problemas que gradativamente vão aumentando de complexidade, requerendo maiores conhecimentos e habilidades.

Entretanto, chega-se a um ponto em que, dado o tamanho ou a complexidade do problema que se pretende resolver, essa abordagem individual, centrada na programação não é mais indicada. De fato, ela só é aplicável para resolver pequenos problemas, tais como calcular médias, ordenar conjuntos de dados etc, envolvendo basicamente o projeto de um único algoritmo. Contudo, é insuficiente para problemas grandes e complexos, tais como aqueles tratados na automação bancária, na informatização de portos ou na gestão empresarial. Em tais situações, uma abordagem de engenharia é necessária.

Observando outras áreas, tal como a Engenharia Civil, podemos verificar que situações análogas ocorrem. Por exemplo, para se construir uma casinha de cachorro, não é necessário elaborar um projeto de engenharia civil, com plantas baixa, hidráulica e elétrica, ou mesmo cálculos estruturais. Um bom pedreiro é capaz de resolver o problema a contento. Talvez não seja dada a melhor solução, mas o produto resultante pode atender aos requisitos pré-estabelecidos. Essa abordagem, contudo, não é viável para a construção de um edifício. Nesse caso, é necessário realizar um estudo aprofundado, incluindo análises de solo, cálculos estruturais etc, seguido de um planejamento da execução da obra e desenvolvimento de modelos (maquetes e plantas de diversas naturezas), até a realização da obra, que deve ocorrer por etapas, tais como fundação, alvenaria, acabamento etc. Ao longo da realização do trabalho, deve-se realizar um acompanhamento para verificar prazos, custos e a qualidade do que se está construindo.

Visando melhorar a qualidade dos produtos de software e aumentar a produtividade no processo de desenvolvimento, surgiu a *Engenharia de Software*. A Engenharia de Software trata de aspectos relacionados ao estabelecimento de processos, métodos, técnicas, ferramentas e ambientes de suporte ao desenvolvimento de software.

Assim como em outras áreas, em uma abordagem de engenharia de software, inicialmente o problema a ser tratado deve ser analisado e decomposto em partes menores, em uma abordagem "dividir para conquistar". Para cada uma dessas partes, uma solução deve ser elaborada. Solucionados os sub-problemas isoladamente, é necessário integrar as soluções. Para tal, uma arquitetura deve ser estabelecida. Para apoiar a resolução de problemas, procedimentos (métodos, técnicas, roteiros etc) devem ser utilizados, bem como ferramentas para parcialmente automatizar o trabalho.

Neste cenário, muitas vezes não é possível conduzir o desenvolvimento de software de maneira individual. Pessoas têm de trabalhar em equipes, o esforço tem de ser planejado, coordenado e acompanhado, bem como a qualidade do que se está produzindo tem de ser sistematicamente avaliada.

1.1 – Qualidade de Software

Uma vez que um dos objetivos da Engenharia de Software é melhorar a qualidade dos produtos de software desenvolvidos, uma questão deve ser analisada: O que é qualidade de software?

Se perguntarmos a um usuário, provavelmente, ele dirá que um produto de software é de boa qualidade se ele satisfizer suas necessidades, sendo fácil de usar, eficiente e confiável. Essa é uma perspectiva externa de observação pelo uso do produto. Por outro lado, para um desenvolvedor, um produto de boa qualidade tem de ser fácil de manter, sendo o produto de software observado por uma perspectiva interna. Já para um cliente, o produto de software deve agregar valor a seu negócio (qualidade em uso).

Em última instância, podemos perceber que a qualidade é um conceito com múltiplas facetas (perspectivas de usuário, desenvolvedor e cliente) e que envolve diferentes características (por exemplo, usabilidade, confiabilidade, eficiência, manutenibilidade, portabilidade, segurança, produtividade) que devem ser alcançadas em níveis diferentes, dependendo do propósito do software. Por exemplo, um sistema de tráfego aéreo tem de ser muito mais eficiente e confiável do que um editor de textos. Por outro lado, um software educacional a ser usado por crianças deve primar muito mais pela usabilidade do que um sistema de venda de passagens aéreas a ser operado por agentes de turismo especializados.

O que há de comum nas várias perspectivas discutidas acima é que todas elas estão focadas no produto de software. Ou seja, estamos falando de qualidade do produto. Entretanto, como garantir que o produto final de software apresenta essas características? Apenas avaliar se o produto final as apresenta é uma abordagem indesejável para o pessoal de desenvolvimento de software, tendo em vista que a constatação *a posteriori* de que o software não apresenta a qualidade desejada pode implicar na necessidade de refazer grande parte do trabalho. É necessário, pois, que a qualidade seja incorporada ao produto ao longo de seu processo de desenvolvimento. De fato, a qualidade dos produtos de software depende fortemente da qualidade dos processos usados para desenvolvê-los e mantê-los.

Seguindo uma tendência de outros setores, a qualidade do processo de software tem sido apontada como fundamental para a obtenção da qualidade do produto. Abordagens de qualidade de processo, tal como a série de padrões ISO 9000, sugerem que melhorando a qualidade do processo de software, é possível melhorar a qualidade dos produtos resultantes. A premissa por detrás dessa afirmativa é a de que processos bem estabelecidos, que incorporam mecanismos sistemáticos para acompanhar o desenvolvimento e avaliar a qualidade, no geral, conduzem a produtos de qualidade. Por exemplo, quando se diz que um fabricante de eletrodomésticos é uma empresa certificada ISO 9001 (uma das normas da série ISO 9000), não se está garantindo que todos os eletrodomésticos por ele produzidos são produtos de qualidade. Mas sim que ele tem um bom processo produtivo, o que deve levar a produtos de qualidade.

Um processo de software, em uma abordagem de Engenharia de Software, envolve diversas atividades que podem ser classificadas quanto ao seu propósito em:

- Atividades de Desenvolvimento (ou Técnicas ou de Construção): são as atividades diretamente relacionadas ao processo de desenvolvimento do software, ou seja, que contribuem diretamente para o desenvolvimento do produto de software a ser

entregue ao cliente. São exemplos de atividades de desenvolvimento: especificação e análise de requisitos, projeto e implementação.

- Atividades de Gerência: são aquelas relacionadas ao planejamento e acompanhamento gerencial do projeto, tais como realização de estimativas, elaboração de cronogramas, análise dos riscos do projeto etc.
- Atividades de Garantia da Qualidade: são aquelas relacionadas com a garantia da qualidade do produto em desenvolvimento e do processo de software utilizado, tais como revisões e inspeções de produtos (intermediários ou finais) do desenvolvimento.

As atividades de desenvolvimento formam a espinha dorsal do desenvolvimento e, de maneira geral, são realizadas segundo uma ordem estabelecida no planejamento. As atividades de gerência e de controle da qualidade são, muitas vezes, ditas atividades de apoio, pois não estão ligadas diretamente à construção do produto final: o software a ser entregue para o cliente, incluindo toda a documentação necessária. Essas atividades, normalmente, são realizadas ao longo de todo o ciclo de vida, sempre que necessário ou em pontos pré-estabelecidos durante o planejamento, ditos marcos ou pontos de controle. A figura 1.1 mostra a relação entre esses tipos de atividades.

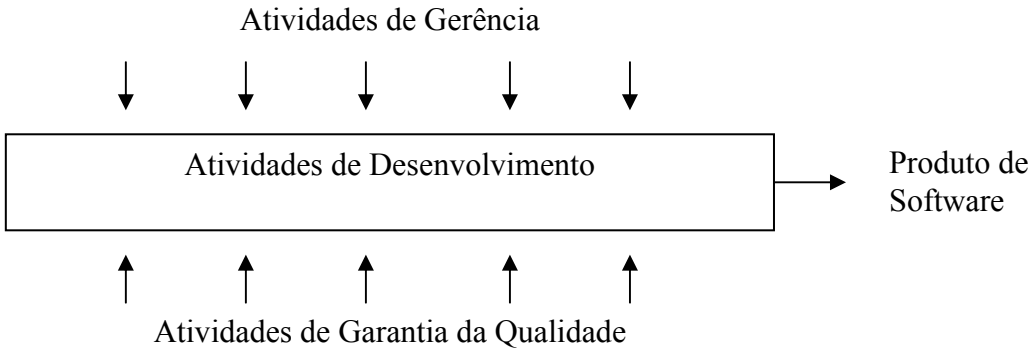


Figura 1.1 – Atividades do Processo de Software

1.2 – A Organização deste Texto

Nesta disciplina, procuramos oferecer uma visão geral da Engenharia de Software, discutindo as principais atividades do processo e como realizá-las. Nos capítulos que se seguem, os seguintes temas são abordados:

- Capítulo 2 – *Processo de Software* – enfoca os processos de software, os elementos que compõem um processo, a definição de processos para projetos, modelos de processo, normas e modelos de qualidade de processo de software e a automatização do processo de software.
- Capítulo 3 – *Planejamento e Gerência de Projetos* – são abordadas as principais atividades da gerência de projetos, a saber: definição do escopo do projeto, estimativas, análise de riscos, elaboração de cronograma, elaboração do plano de projeto e acompanhamento de projetos.

- Capítulo 4 – *Gerência da Qualidade* – trata das principais atividades de garantia da qualidade, incluindo a medição e métricas associadas, revisões e inspeções e a gerência de configuração de software.
- Capítulo 5 – *Especificação e Análise de Requisitos* – são discutidos o que é um requisito de software e tipos de requisitos. A seguir, são abordadas a especificação e a análise de requisitos, usando o método da Análise Essencial de Sistemas como base.
- Capítulo 6 – *Projeto de Sistema* – aborda os conceitos básicos de projeto de sistemas, tratando da arquitetura do sistema a ser desenvolvido e do projeto de seus módulos, segundo a abordagem do Projeto Estruturado de Sistemas.
- Capítulo 7 – *Implementação e Testes* – são enfocadas as atividades de implementação e testes, sendo esta última tratada em diferentes níveis, a saber: teste de unidade, teste de integração, teste de validação e teste de sistema.
- Capítulo 8 – *Entrega e Manutenção* – discute as questões relacionadas à entrega do sistema para o cliente, tais como o treinamento e a documentação de entrega, e a atividade de manutenção do sistema.

Referências para o Conteúdo do Capítulo

- S.L. Pfleeger, *Engenharia de Software: Teoria e Prática*, São Paulo: Prentice Hall, 2ª edição, 2004. Capítulo 1.

2 – Processo de Software

O que é um processo de software.

Um processo de software pode ser visto como o conjunto de atividades, métodos, práticas e transformações que guiam pessoas na produção de software. Um processo eficaz deve, claramente, considerar as relações entre as atividades, os artefatos produzidos no desenvolvimento, as ferramentas e os procedimentos necessários e a habilidade, o treinamento e a motivação do pessoal envolvido.

Elementos que compõem um processo de software

- Processo de Software
  - Processos
    - Atividades
      - Pré-atividades
      - Sub-atividades
      - Artefatos
        - Insumos
        - Produtos
      - Recursos
        - Recursos Humanos
        - Ferramentas de Software
        - Hardware
      - Procedimentos
        - Métodos
        - Técnicas
        - Roteiros

Definição de Processos

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

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

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

A escolha de um modelo de ciclo de vida (ou modelo de processo) é o ponto de partida para a definição de um processo de desenvolvimento de software. Um modelo de ciclo de

vida organiza as macro-atividades básicas, estabelecendo precedência e dependência entre as mesmas. Para maiores detalhes sobre os modelos de processo existentes, vide as seguintes referências sugeridas: [1,2,3].

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

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

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

componentes devem ser integrados sucessivamente até se obter o sistema. Finalmente, o sistema como um todo deve ser testado.

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

São fatores que influenciam a definição de um processo: Tipo de Software (p.ex., sistema de informação, sistema de tempo real etc), Paradigma (estruturado, orientado a objetos etc), Domínio da Aplicação, Tamanho e Complexidade, Características da Equipe etc.

Embora diferentes projetos requeiram processos com características específicas para atender às suas particularidades, é possível estabelecer um conjunto de ativos de processo (sub-processos, atividades, sub-atividades, artefatos, recursos e procedimentos) a ser utilizado na definição de processos de software de uma organização. Essas coleções de ativos de processo de software constituem os chamados processos padrão de desenvolvimento de software. Processos para projetos específicos podem, então, ser definidos a partir da instanciação do processo de software padrão da organização, levando em consideração suas características particulares. Esses processos instanciados são ditos processos de projeto.

De fato, o modelo de definição de processos baseado em processos padrão pode ser estendido para comportar vários níveis. Primeiro, pode-se definir um processo padrão da organização, contendo os ativos de processo que devem fazer parte de **todos** os processos de projeto da organização. Esse processo padrão pode ser especializado para agregar novos ativos de processo, considerando aspectos, tais como tecnologias de desenvolvimento, paradigmas ou domínios de aplicação. Assim, obtêm-se processos mais completos, que consideram características da especialização desejada. Por fim, a partir de um processo padrão ou de um processo especializado, é possível instanciar um processo de projeto, que será o processo a ser utilizado em um projeto de software específico. Para definir esse processo devem ser consideradas as particularidades de cada projeto [4].

Para apoiar a definição de processos, diversas normas e modelos de qualidade de processo de software foram propostas, dentre elas [4]: ISO 9001, ISO/IEC 12207, ISO/IEC 15504, CMM e CMMI. O objetivo dessas normas e modelos de qualidade é apontar características que um bom processo de software tem de apresentar, deixando a organização livre para estruturar essas características segundo sua própria cultura.

Assim, usando essas normas e modelos de qualidade, em uma abordagem de definição de processos em níveis, é possível definir processos para projetos específicos, que levem em consideração as particularidades de cada projeto, sem, no entanto, desconsiderar aspectos importantes para se atingir a qualidade do processo. A figura 2.1 ilustra essa abordagem de definição de processos de software em níveis [4].

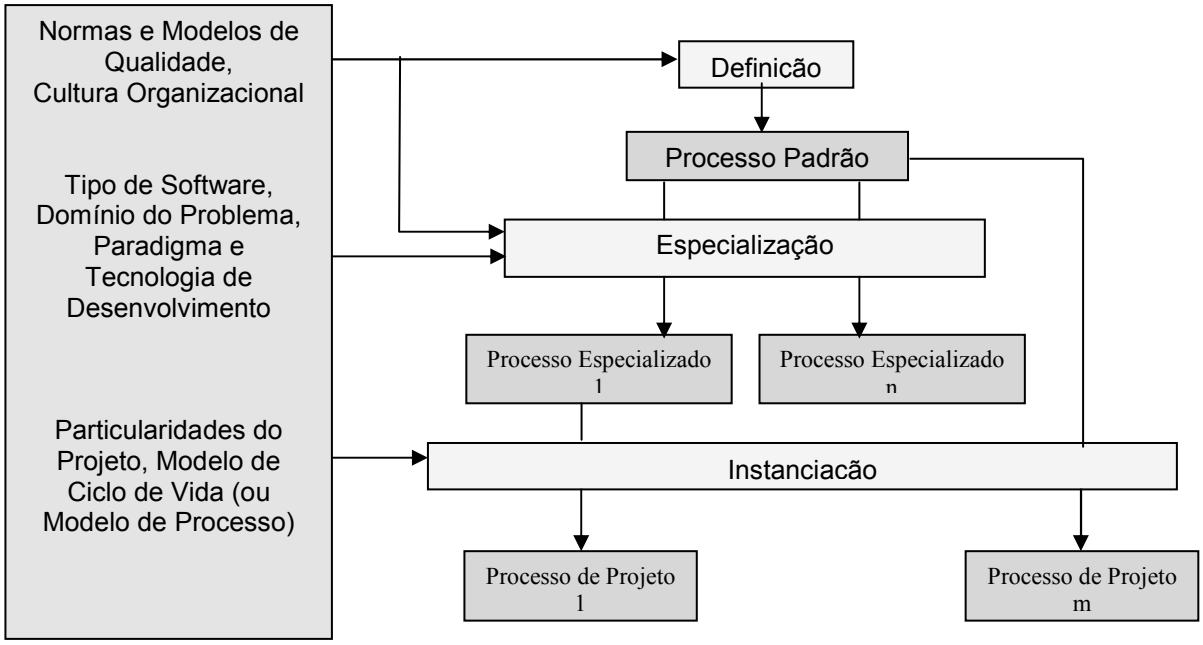


Figura 2.1 – Modelo para Definição de Processos em Níveis

Automatização do Processo de Software [1]: Capítulo 31, [2] Capítulo 3, seção 3.7

Com o aumento da complexidade dos processos de software, passou a ser imprescindível o uso de ferramentas e ambientes de apoio à realização de suas atividades, visando, sobretudo, a atingir níveis mais altos de qualidade e produtividade. Ferramentas CASE (*Computer Aided Software Engineering*) passaram, então, a ser utilizadas para apoiar a realização de atividades específicas, tais como planejamento e análise e especificação de requisitos [1].

Apesar dos benefícios do uso de ferramentas CASE individuais, atualmente, o número e a variedade de ferramentas têm crescido a tal ponto que levou os engenheiros de software a pensarem não apenas em automatizar os seus processos, mas sim em trabalhar com diversas ferramentas que interajam entre si e forneçam suporte a todo ciclo de vida do desenvolvimento, dando origem ao Ambientes de Desenvolvimento de Software (ADSs).

ADSs buscam combinar técnicas, métodos e ferramentas para apoiar o engenheiro de software na construção de produtos de software, abrangendo todas as atividades inerentes ao processo: gerência, desenvolvimento e controle da qualidade.

Referências para o Conteúdo do Capítulo

1. R.S. Pressman, *Engenharia de Software*, Rio de Janeiro: McGraw Hill, 5ª edição, 2002: Capítulos 2 (Processo de Software) e 31 (Engenharia de Software Apoiada por Computador).
2. I. Sommerville, *Engenharia de Software*, São Paulo: Addison-Wesley, 6ª edição, 2003: Capítulo 3.
3. S.L. Pfleeger, *Engenharia de Software: Teoria e Prática*, São Paulo: Prentice Hall, 2ª edição, 2004. Capítulo 2.
4. A. R. C. Rocha, J. C. Maldonado, K. C. Weber, *Qualidade de Software: Teoria e Prática*, São Paulo: Prentice Hall, 2001. [1]: Capítulos 1 (Normas e Modelos de Qualidade de Processo) e 12 (Automatização da Definição de Processos de Software).

3 – Gerência de Projetos de Software

A Gerência de Projetos de Software envolve o Produto, o Processo e as Pessoas envolvidas no projeto.

Produto

Referências: [1]: Cap. 3, seção 3.3, Cap. 5, seção 5.3; [3]: Cap. 3, seção 3.1.

No que se refere ao produto, a primeira coisa a se fazer é definir o escopo do projeto. Para tal, é necessário fazer um levantamento de requisitos inicial. A idéia é decompor o problema, em uma abordagem "dividir para conquistar". Inicialmente, o sistema deve ser decomposto em subsistemas que são, por sua vez, decompostos em módulos. Os módulos podem, ainda, ser recursivamente decompostos em sub-módulos ou funções, até que se tenha uma visão geral das funcionalidades a serem tratadas no projeto.

Processo: já estudado no capítulo anterior

Pessoas:

Referências: [1] Cap. 3, seção 3.2; [2] Cap. 22; [3] Cap. 3, seção 3.2.

Em um projeto de software, há várias pessoas envolvidas, exercendo diferentes papéis, tais como: Gerente de Projeto, Desenvolvedor (Analistas, Projetistas, Engenheiro de Software, Programadores, Engenheiros de Testes), Gerente da Qualidade, Clientes, Usuários. O número de papéis e suas denominações podem ser bastante diferentes dependendo da organização e até mesmo do projeto.

As pessoas trabalhando em um projeto são organizadas em equipes. Assim, o conceito de equipe pode ser visto como um conjunto de pessoas trabalhando em diferentes tarefas, mas objetivando uma meta comum. Essa não é uma característica do desenvolvimento de software, mas da organização de pessoas em qualquer atividade humana. Assim, a definição de equipe dada é válida para uma ampla variedade de situações, tal como uma equipe de futebol.

Para a boa formação de equipes, devem ser definidos os papéis necessários e devem ser considerados aspectos fundamentais, a saber liderança, organização (estrutura da equipe) e coordenação. Além disso, há diversos fatores que afetam a formação de equipes: relacionamentos inter-pessoais, tipo do projeto, criatividade etc.

No que se refere à organização / estrutura das equipes, há diversos tipos de equipes, tais como os citados por Pressman [1]: Democrática Descentralizada, Controlada Descentralizada, Controlada Centralizada.

Por fim, na formação de equipes deve-se levar em conta o tamanho da equipe. Quanto maior o número de membros da equipe, maior a quantidade de caminhos possíveis de comunicação, o que pode ser um problema, uma vez que o número de pessoas que podem se comunicar com outras pode afetar a qualidade do produto resultante.

Estrutura de Divisão do Trabalho:

Referências: [1] Cap. 3, seção 3.4; [2] Cap. 2, seção 2.3; [3] Cap. 3, seção 3.1.

Uma boa gerência de projetos começa com a fusão das visões de Produto, Processo e Pessoas. Cada função ou módulo a ser desenvolvido pela equipe do projeto deve passar pelas várias atividades definidas no processo de software. Essa pode ser uma base bastante efetiva para a elaboração de estimativas, incluindo a alocação de recursos, já que é sempre mais fácil estimar porções menores de trabalho. Assim, é útil elaborar uma estrutura de divisão do trabalho, considerando duas dimensões principais: produto e processo, como mostra a Tabela 3.1.

Tabela 3.1 – Estrutura de Divisão do Trabalho considerando a fusão das visões de produto e processo.

	Atividades do processo			
	Análise e Especificação de Requisitos	Projeto	Implementação	Testes
Módulos / Funções				
Módulo 1				
Módulo 2				
....				

Atividades Típicas da Gerência de Projetos:

A gerência de projetos envolve a realização de diversas atividades, abaixo relacionadas:

Determinação do Escopo do Software

Definição do Processo de Software do Projeto

Realização de Estimativas

- Estimativa de Tamanho
- Estimativa de Esforço
- Estimativa (Alocação) de Recursos
- Estimativa de Tempo (Elaboração do Cronograma do Projeto)
- Estimativa de Custos

Gerência de Riscos

Elaboração do Plano de Projeto

O Planejamento e o Acompanhamento do Projeto

As atividades acima relacionadas são realizadas diversas vezes ao longo do projeto. Tipicamente, no início do projeto, elas têm de ser realizadas para produzir uma primeira visão

gerencial sobre o projeto, quando são conjuntamente denominadas de planejamento do projeto. À medida que o projeto avança, contudo, o plano do projeto deve ser revisto, uma vez que problemas podem surgir ou porque se ganha um maior entendimento sobre o problema. Essas revisões do plano de projeto são ditas atividades de acompanhamento do projeto e tipicamente são realizadas nos marcos do projeto.

Os marcos de um projeto são estabelecidos durante a definição do processo e tipicamente correspondem ao término de atividades importantes do processo de desenvolvimento, tais como Análise e Especificação de Requisitos, Projeto e Implementação. O propósito de um marco é garantir que os interessados tenham uma visão do andamento do projeto e concordem com os rumos a serem tomados.

Em uma atividade de acompanhamento do projeto, o escopo pode ser revisto, alterações no processo podem ser necessárias, bem como devem ser monitorados os riscos e revisadas as estimativas (de tamanho, esforço, tempo e custo).

3.1 - Estimativas

Referências: [1] Cap. 5; [2] Cap. 23; [3] Cap. 3, seção 3.3.

Antes mesmo de serem iniciadas as atividades técnicas de um projeto, o gerente e a equipe de desenvolvimento devem estimar o trabalho a ser realizado, os recursos necessários, o tempo de duração e, por fim, o custo do projeto. Apesar das estimativas serem um pouco de arte e um pouco de ciência, esta importante atividade não deve ser conduzida desordenadamente. As estimativas podem ser consideradas a fundação para todas as outras atividades de planejamento de projeto. Para alcançar boas estimativas de prazo, esforço e custo, existem algumas opções [1]:

1. Postergar as estimativas até o mais tarde possível no projeto.
2. Usar técnicas de decomposição.
3. Usar um ou mais modelos empíricos para estimativas de custo e esforço.
4. Basear as estimativas em projetos similares que já tenham sido concluídos.

A primeira opção, apesar de ser atraente, não é prática, pois estimativas devem ser providas logo no início do projeto (fase de planejamento do projeto). No entanto, deve-se reconhecer que quanto mais tarde for feita a estimativa, maior o conhecimento do projeto e menores as chances de se cometer erros. Assim, é fundamental revisar as estimativas na medida em que o projeto avança (atividades de acompanhamento do projeto).

Técnicas de decomposição, a segunda opção, usam, conforme discutido anteriormente, a abordagem “dividir para conquistar” na realização de estimativas, através da decomposição do projeto em módulos / funções (decomposição do produto) e atividades mais importantes (decomposição do processo). Assim, uma tabela como a Tabela 3.1 pode ser utilizada para estimar, por exemplo, tamanho ou esforço.

Modelos empíricos, tipicamente, usam fórmulas matemáticas, derivadas em experimentos, para prever esforço como uma função de tamanho (linhas de código ou pontos de função). Entretanto, deve-se observar que os dados empíricos que suportam a maioria desses modelos são derivados de um conjunto limitado de projetos. Além disso, fatores culturais da organização não são considerados no uso de modelos empíricos, pois os projetos que constituem a base de dados do modelo são externos à organização. Apesar de suas

limitações, modelos empíricos podem ser úteis como um ponto de partida para organizações que ainda não têm dados históricos, até que a organização possa estabelecer suas próprias correlações.

Finalmente, na última opção, dados de projetos anteriores armazenados em um repositório de experiências da organização podem prover uma perspectiva histórica importante e ser uma boa fonte para estimativas. Através de mecanismos de busca, é possível recuperar projetos similares, suas estimativas e lições aprendidas, que podem ajudar a elaborar estimativas mais precisas. Nesta abordagem, os fatores culturais são considerados, pois os projetos foram desenvolvidos na própria organização.

Vale frisar que essas abordagens não são excludentes; muito pelo contrário. O objetivo é ter várias formas para realizar estimativas e usar seus resultados para se chegar a estimativas mais precisas.

Quando se fala em estimativas, está-se tratando na realidade de diversos tipos de estimativas: tamanho, esforço, recursos, tempo e custos. Geralmente, a realização de estimativas começa pelas estimativas de tamanho. A partir delas, estima-se o esforço necessário e, na seqüência, alocam-se os recursos necessários, elabora-se o cronograma do projeto (estimativas de tempo) e, por fim, estima-se o custo do projeto.

Estimativa de Tamanho:

Referências: [1] Cap. 5, seção 5.6; [2] Cap. 23, seção 23.1; [3] Cap. 3, seção 3.3.

Entre as várias formas de se medir tamanho de um software, a mais simples, direta e altamente utilizada é a contagem do número de linhas de código (*Lines Of Code* - LOC) dos programas fonte. Existem alguns estudos que demonstram a alta correlação entre essa métrica e o tempo necessário para se desenvolver um sistema. Entretanto, o uso dessa métrica apresenta algumas desvantagens. Primeiro, verifica-se que ela é fortemente ligada à tecnologia empregada, sobretudo a linguagem de programação. Segundo, pode ser difícil estimar essa grandeza no início do desenvolvimento, sobretudo se não houver dados históricos relacionados com a linguagem de programação utilizada no projeto.

Visando possibilitar a realização de estimativas de tamanho mais cedo no processo de software, foram propostas outras métricas em um nível de abstração mais alto. O exemplo mais conhecido é a contagem de Pontos de Função (PFs), que busca medir as funcionalidades do sistema requisitadas e recebidas pelo usuário, de forma independente da tecnologia usada na implementação. Seu maior problema é que os dados necessários para a Análise de PFs são bastante imprecisos no início de um projeto e, portanto, gerentes de projeto são, muitas vezes, obrigados a produzir estimativas antes de um estudo mais aprofundado. Assim, os pontos de função devem ser recontados ao longo do processo (nas atividades de acompanhamento de projetos), para que ajustes de previsões possam ser realizados e controlados, fornecendo *feedback* para situações futuras.

Muitas organizações coletam dados para permitir a conversão de PFs em LOCs. Quando não há dados organizacionais para se fazer essa conversão, podem ser utilizados dados gerais reportados na literatura, como os providos em [4]. Nesse trabalho, uma lista de valores de LOCs por PFs é fornecida para diversas linguagens, que, na média, apresenta valores como os mostrados na Tabela 3.2.

Tabela 3.2 – Relação Número Médio de LOC / PF para algumas Linguagens de Programação

Linguagem de Programação	Número Médio de LOC/PF
C	162
C++	66
Java	63
SQL	40

Estimativas de Esforço:

Referências: [1] Cap. 5, seções 5.6 e 5.7; [2] Cap. 23, seção 23.1; [3] Cap. 3, seção 3.3.

Para a realização de estimativas de tempo e custo, é fundamental estimar, antes, o esforço necessário para completar o projeto ou cada uma de suas atividades. Estimativas de esforço podem ser obtidas diretamente pelo julgamento de especialistas, tipicamente usando técnicas de decomposição, ou podem ser computadas a partir de dados de tamanho ou de dados históricos.

Quando as estimativas de esforço são feitas com base apenas no julgamento dos especialistas, uma tabela como a Tabela 3.1 pode ser utilizada, em que cada célula corresponde ao esforço necessário para efetuar uma atividade no escopo de um módulo específico. Uma tabela como essa pode ser produzida também com base em dados históricos de projetos similares já realizados na organização.

Quando estimativas de tamanho são usadas como base, deve-se considerar um fator de produtividade, indicando quanto em unidades de esforço é necessário para completar um projeto (ou módulo), descrito em unidades de tamanho. Assim, uma organização pode coletar dados de vários projetos e estabelecer, por exemplo, quantos em homens-hora (uma unidade de esforço) são necessários para desenvolver 1000 LOCs (KLOC) ou 1 PF (unidades de tamanho). Esses fatores de produtividade devem levar em conta características dos projetos e da organização. Assim, pode ser útil ter vários fatores de produtividade, considerando classes de projetos específicas.

Assim como em outras situações, quando uma organização não tem ainda dados suficientes para definir seus próprios fatores de produtividade, modelos empíricos podem ser usados. Existem diversos modelos que derivam estimativas de esforço a partir de dados de LOC ou PFs. De maneira geral, todos eles têm a seguinte estrutura:

$$E = A + B * (T)^c$$

onde *A*, *B* e *C* são constantes derivadas empiricamente, *E* é o esforço em pessoas-mês e *T* é a estimativa de tamanho em LOCs ou PFs.

Por exemplo, o modelo proposto por Bailey-Basili estabelece a seguinte fórmula para se obter o esforço necessário em pessoas-mês para desenvolver um projeto, tomando por base o tamanho do mesmo em KLOC:

$$E = 5,5 + 0,73 * (KLOC)^{1,16}$$

Segundo os critérios citados por Pfleeger [3], esse é um dos modelos empíricos mais precisos. Contudo, deve-se observar que modelos empíricos diferentes conduzem a resultados muito diferentes, o que indica que esses modelos devem ser adaptados para as condições da organização. Uma forma de se fazer essa adaptação consiste em experimentar o modelo usando resultados de projetos já finalizados, comparar os valores obtidos com os dados reais e analisar a eficácia do modelo. Se a concordância dos resultados não for boa, as constantes do modelo devem ser recalculadas usando dados organizacionais [1].

Alocação de Recursos:

Referências: [1] Cap. 5, seção 5.4; [2] Cap. 22, seção 22.3;

Estimar os recursos necessários para realizar o esforço de desenvolvimento é outra importante tarefa. Quando falamos em recursos, estamos englobando pessoas, hardware e software. No caso de software, devemos pensar em ferramentas de software, tais como ferramentas CASE ou software de infra-estrutura (p.ex., um sistema operacional), bem como componentes de software a serem reutilizados no desenvolvimento, tais como bibliotecas de interface ou uma camada de persistência de dados.

Em todos os casos (recursos humanos, de hardware e de software), é necessário observar a disponibilidade do recurso. Assim, é importante definir a partir de que data o recurso será necessário, por quanto tempo ele será necessário e qual a quantidade de horas necessárias por dia nesse período, o que, para recursos humanos, convencionamos denominar dedicação. Observe que já entramos na estimativa de tempo. Assim, alocação de recursos e estimativa de tempo são atividades realizadas normalmente em paralelo.

No que se refere a recursos humanos, outros fatores têm de ser levados em conta. A competência para realizar a atividade para a qual está sendo alocado é fundamental. Assim, é preciso analisar com cuidado as competências dos membros da equipe para poder realizar a alocação de recursos. Outros fatores, como liderança, relacionamento inter-pessoal etc, importantes para a formação de equipes, são igualmente relevantes para a alocação de recursos humanos a atividades.

Estimativa de Tempo:

Referências: [1] Cap. 7; [2] Cap. 4, seção 4.3; [3] Cap. 3, seção 3.1.

De posse das estimativas de esforço e realizando em paralelo a alocação de recursos, é possível estimar o tempo de cada atividade e, por conseguinte, do projeto. Se a estimativa de esforço tiver sido realizada para o projeto como um todo, então ela deverá ser distribuída pelas atividades do projeto. Novamente, dados históricos de projetos já concluídos na organização são uma boa base para se fazer essa distribuição.

No entanto, muitas vezes, uma organização não tem ainda esses dados disponíveis. Embora as características do projeto sejam determinantes para a distribuição do esforço, uma diretriz inicial útil consiste em considerar a distribuição mostrada na Tabela 3.3 [1].

Tabela 3.3 – Distribuição de Esforço pelas Principais Atividades do Processo de Software.

Planejamento	Especificação e Análise de Requisitos	Projeto	Implementação	Teste e Entrega
Até 3%	De 10 a 25%	De 20 a 25%	De 15 a 20%	De 30 a 40%

De posse da distribuição de esforço por atividade e realizando paralelamente a alocação de recursos, pode-se criar uma rede de tarefas com o esforço associado a cada uma das atividades. A partir dessa rede, pode-se estabelecer qual é o caminho crítico do projeto, isto é, qual o conjunto de atividades que determina a duração do projeto. Um atraso em uma dessas atividades provocará atraso no projeto como um todo.

Finalmente, a partir da rede de tarefas, deve-se elaborar um Gráfico de Tempo (ou Gráfico de Gantt), estabelecendo o cronograma do projeto. Gráficos de Tempo podem ser elaborados para o projeto como um todo (cronograma do projeto), para um conjunto de atividades, para um módulo específico ou mesmo para um membro da equipe do projeto.

Estimativa de Custo:

Referências: [1] Cap. 5, seções 5.6 e 5.7; [2] Cap. 23.

De posse das demais estimativas, é possível estimar os custos do projeto. De maneira geral, os seguintes itens devem ser considerados nas estimativas de custos:

- Custos relativos ao esforço empregado pelos membros da equipe no projeto;
- Custos de hardware e software (incluindo manutenção);
- Outros custos relacionados ao projeto, tais como custos de viagens e treinamentos realizados no âmbito do projeto;
- Despesas gerais, incluindo gastos com água, luz, telefone, pessoal de apoio administrativo, pessoal de suporte etc.

Para a maioria dos projetos, o custo dominante é o que se refere ao esforço empregado, juntamente com as despesas gerais. Sommerville [2] sugere que, de modo geral, os custos relacionados com as despesas gerais correspondem a um valor equivalente aos custos relativos ao esforço empregado pelos membros da equipe no projeto. Assim, para efeitos de estimativas de custos, pode-se considerar esses dois itens como sendo um único, computado em dobro.

Custos de hardware e software, ainda que menos influentes, não devem ser desconsiderados, sob pena de provocarem prejuízos para o projeto. Uma forma de tratar esses custos é considerar a depreciação com base na vida útil do equipamento ou da versão do software utilizada.

Quando o custo do projeto estiver sendo calculado como parte de uma proposta para o cliente, então será preciso definir o preço cotado. Uma abordagem para definição do preço pode ser considerá-lo como o custo total do projeto mais o lucro. Entretanto, a relação entre o custo do projeto e o preço cotado para o cliente, normalmente, não é tão simples assim [2].

3.2 - Gerência de Riscos

Referências: [1] Cap. 6; [2] Cap. 4, seção 4.4; [3] Cap. 3, seção 3.4.

Uma importante tarefa da gerência de projetos é prever os riscos que podem prejudicar o bom andamento do projeto e definir ações a serem tomadas para evitar sua ocorrência ou, quando não for possível evitar a ocorrência, para diminuir seus impactos.

Um risco é qualquer condição, evento ou problema cuja ocorrência não é certa, mas que pode afetar negativamente o projeto, caso ocorra. Assim, os riscos envolvem duas características principais:

- *incerteza* – um risco pode ou não acontecer, isto é, não existe nenhum risco 100% provável;
- *perda* – se o risco se tornar realidade, conseqüências não desejadas ou perdas acontecerão.

Desta forma, é de suma importância que riscos sejam identificados durante um projeto de software, para que ações possam ser planejadas e utilizadas para evitar que um risco se torne real, ou para minimizar seus impactos, caso ele ocorra. Esse é o objetivo da Gerência de Riscos, cujo processo envolve as seguintes atividades:

- Identificação de riscos: visa identificar possíveis ameaças (riscos) para o projeto, antecipando o que pode dar errado;
- Análise de riscos: trata de analisar os riscos identificados, estimando sua probabilidade e impacto (grau de exposição ao risco);
- Avaliação de riscos: busca priorizar os riscos e estabelecer um ponto de corte, indicando quais riscos serão gerenciados e quais não serão;
- Planejamento de ações: trata do planejamento das ações a serem tomadas para evitar (ações de mitigação) que um risco ocorra ou para definir o que fazer quando um risco se tornar realidade (ações de contingência);
- Elaboração do Plano de Riscos: todos os aspectos envolvidos na gerência de riscos devem ser documentados em um plano de riscos, indicando os riscos que compõem o perfil de riscos do projeto, as avaliações dos riscos, a definição dos riscos a serem gerenciados e, para esses, as ações para evitá-los ou para minimizar seus impactos, caso venham a ocorrer.
- Monitoramento de riscos: à medida que o projeto progride, os riscos têm de ser monitorados para se verificar se os mesmos estão se tornando realidade ou não. Novos riscos podem ser identificados, o grau de exposição de um risco pode mudar e ações podem ter de ser tomadas. Essa atividade é realizada durante o acompanhamento do progresso do projeto.

Na identificação de riscos, trabalhar com uma série de riscos aleatórios pode ser um fator complicador, principalmente em grandes projetos, em que o número de riscos é relativamente grande. Assim, a classificação de riscos em categorias de risco, definindo tipos básicos de riscos, é importante para guiar a realização dessa atividade. Cada organização deve ter seu próprio conjunto de categorias de riscos. Para efeito de exemplo, podem ser consideradas categorias tais como: tecnologia, pessoal, legal, organizacional, de negócio etc.

Uma vez identificados os riscos, deve ser feita uma análise dos mesmos à luz de suas duas principais variáveis: a probabilidade do risco se tornar real e o impacto do mesmo, caso

ocorra. Na análise de riscos, o gerente de projeto deve executar quatro atividades básicas [1]: (i) estabelecer uma escala que reflita a probabilidade de um risco, (ii) avaliar as conseqüências dos riscos, (iii) estimar o impacto do risco no projeto e no produto e (iv) calcular o grau de exposição do risco, que é uma medida casando probabilidade e impacto.

De maneira geral, escalas para probabilidades e impactos são definidas de forma qualitativa, tais como: probabilidade - alta, média ou baixa, e impacto - baixo, médio, alto ou muito alto. Isso facilita a análise dos riscos, mas, por outro lado, pode dificultar a avaliação. Assim, a definição de medidas quantitativas para o risco pode ser importante, pois tende a diminuir a subjetividade na avaliação. Jalote [5] propõe os valores quantitativos mostrados nas tabelas 3.4 e 3.5 para as escalas acima.

Tabela 3.4 - Categorias de Probabilidade [5]

Probabilidade	Faixa de Valores
Baixa	até 30%
Média	30 a 70%
Alta	acima de 70%

Tabela 3.5 - Categorias de Impacto [5]

Impacto	Faixa de Valores
Baixo	de 0 a 3
Médio	de 3 a 7
Alto	de 7 a 9
Muito Alto	de 9 a 10

Usando valores quantitativos para expressar probabilidade e impacto, é possível obter um valor numérico para o grau de exposição ao risco, dado por: $E(r) = P(r) * I(r)$, onde $E(r)$ é o grau de exposição associado ao risco r e $P(r)$ e $I(r)$ correspondem, respectivamente, aos valores numéricos de probabilidade e impacto do risco r.

De posse do grau de exposição de cada um dos riscos que compõem o perfil de riscos do projeto, pode-se passar à avaliação dos mesmos. Uma tabela ordenada pelo grau de exposição pode ser montada e uma linha de corte nessa tabela estabelecida, indicando quais riscos serão tratados e quais serão desprezados.

Para os riscos a serem gerenciados, devem ser definidos planos de ação, estabelecendo ações a serem adotadas para, em primeiro lugar, evitar que os riscos ocorram (ações de mitigação) ou, caso isso não seja possível, ações para tratar e minimizar seus impactos (ações de contingência). Esse é o propósito da atividade de planejamento das ações.

Ao longo de todo o processo da gerência de riscos, as decisões envolvidas devem ser documentadas no plano de riscos. Esse plano servirá de base para a atividade de monitoramento dos riscos, quando os riscos têm de ser monitorados para se verificar se os mesmos estão se tornando realidade ou não. Caso estejam se tornando (ou já sejam) uma

realidade, deve-se informar que ações foram tomadas. No caso do risco se concretizar, deve-se informar, também, quais as conseqüências da sua ocorrência.

3.3 - Elaboração do Plano de Projeto

Referências: [1] Cap. 7, seção 7.10; [2] Cap. 4, seção 4.2; [3] Cap. 3, seção 3.5.

Todas as atividades realizadas no contexto da gerência de projeto devem ser documentadas em um Plano de Projeto. Cada organização deve estabelecer um modelo ou padrão para a elaboração desse documento, de modo que todos os projetos da organização contenham as informações consideradas relevantes.

Referências

1. R.S. Pressman, *Engenharia de Software*, Rio de Janeiro: McGraw Hill, 5ª edição, 2002.
2. I. Sommerville, *Engenharia de Software*, São Paulo: Addison-Wesley, 6ª edição, 2003.
3. S.L. Pfleeger, *Engenharia de Software: Teoria e Prática*, São Paulo: Prentice Hall, 2ª edição, 2004.
4. "QSM Function Point Language Gearing Factors", Version 2.0, Quantitative Software Management, 2002, http://www.qsm.com/FPGearing.html.
5. P. Jalote, *CMM in Practice: Processes For Executing Software Projects At Infosys*, Addison-Wesley Publishing Company, 1999.

4 – Gerência da Qualidade

Documentação

A documentação produzida em um projeto de software é de suma importância para se gerenciar a qualidade, tanto do produto sendo produzido, quanto do processo usado para seu desenvolvimento.

No desenvolvimento de software, são produzidos diversos documentos, dentre eles, documentos descrevendo processos (plano de projeto, plano de qualidade etc.), registrando requisitos e modelos do sistema (documentos de especificação de requisitos, análise e projeto) e apoiando o uso do sistema gerado (manual do usuário, ajuda *on-line*, tutoriais etc).

Uma documentação de qualidade propicia uma maior organização durante o desenvolvimento de um sistema, facilitando modificações e futuras manutenções no mesmo. Além disso, reduz o impacto da perda de membros da equipe, reduz o tempo de desenvolvimento de fases posteriores, reduz o tempo de manutenção e contribui para redução de erros, aumentando assim, a qualidade do processo e do produto gerado. Dessa forma, a criação da documentação é tão importante quanto a criação do software em si [4].

Há, portanto, a necessidade de se definir um processo para controlar a documentação de uma organização, dito processo de documentação, incluindo atividades de planejamento, análise, aprovação ou reprovação, identificação de alterações, situação da revisão atual, disponibilidade das versões pertinentes de documentos aplicáveis, dentre outras. Algumas dessas atividades estão relacionadas com o controle e a garantia da qualidade de software, outras com a gerência da configuração do software, conforme discutido a seguir.

É importante notar que o planejamento da documentação tem uma estreita relação com o processo de software definido para o projeto. Ou seja, os documentos a serem gerenciados são aqueles previstos como saídas das atividades do processo. Assim, tendo sido definido o processo do projeto, o planejamento da sua documentação consiste apenas em selecionar quais artefatos, dentre os muitos produzidos ao longo do processo, serão efetivamente submetidos à gerência de configuração de software e ao controle e garantia da qualidade.

Controle e Garantia da Qualidade

Referências: [1] Cap. 8, seções 8.3, 8.4 e 8.11; [2] Cap. 24, seções 24.1, 24.2, 24.3;

Durante o processo de desenvolvimento de software, ocorrem enganos, interpretações errôneas e outras faltas (defeitos ou erros), principalmente provocados por problemas na comunicação e transformação da informação, que podem resultar em mau funcionamento do sistema produzido. Assim, é muito importante detectar esses defeitos o quanto antes, preferencialmente na atividade em que foram cometidos, como forma de diminuir retrabalho e, por conseguinte, custos de alterações. As atividades que se preocupam com essa questão são coletivamente denominadas atividades de garantia da qualidade de software e devem ser realizadas ao longo de todo o processo de desenvolvimento de software [5].

Dentre as atividades de controle e garantia da qualidade estão as atividades de Verificação, Validação e Testes (VV&T). O objetivo da verificação é assegurar que o software esteja sendo construído de forma correta. Deve-se verificar se os artefatos produzidos atendem aos requisitos estabelecidos e se os padrões organizacionais (de produto e

processo) foram consistentemente aplicados. Por outro lado, o objetivo da validação é assegurar que o software que está sendo desenvolvido é o software correto, ou seja, se os requisitos e o software dele derivado atendem ao uso específico proposto. Por fim, os testes de software são atividades de validação e verificação que consistem da análise dinâmica do mesmo, isto é, envolvem a execução do produto de software.

Uma vez que as atividades de VV&T são tão importantes, elas devem ser cuidadosamente planejadas, dando origem a um Plano de Garantia da Qualidade.

Os artefatos que compõem a documentação do projeto são as entradas (insumos) para as atividades de garantia da qualidade, quando os mesmos são verificados quanto à aderência em relação aos padrões de documentação da organização e validados em relação aos seus propósitos e aos requisitos que se propõem a atender. Assim, uma questão imprescindível para a gerência da qualidade é a definição de padrões organizacionais.

Padrões Organizacionais

Uma vez que a gerência da qualidade envolve tanto a qualidade do processo quanto a qualidade do produto, devem ser estabelecidos padrões organizacionais de produto e de processo. Os padrões de processo são os ditos processos padrão ou processos padrão especializados, discutidos no capítulo 2.

Os padrões de produto, por sua vez, são padrões que se aplicam a artefatos produzidos ao longo do processo de software. Podem ser, dentre outros, modelos de documentos, roteiros e normas, dependendo do artefato a que se aplicam. Tipicamente, para documentos, modelos de documentos e roteiros são providos.

Um modelo de documento define a estrutura (seções, sub-seções, informações de cabeçalho e rodapé de página etc), o estilo (tamanho e tipos de fonte, cores etc) e o conteúdo esperado para documentos de um tipo específico. Documentos tais como Plano de Projeto, Especificação de Requisitos e Especificação de Projeto devem ter modelos de documentos específicos associados. Documentos padronizados são importantes, pois facilitam a leitura e a compreensão, uma vez que os profissionais envolvidos estão familiarizados com seu formato.

Quando não é possível ou desejável estabelecer uma estrutura rígida como um modelo de documento, roteiros dando diretrizes gerais para a elaboração de um artefato devem ser providos. Em situações onde não é possível definir uma estrutura, tal como no código-fonte, normas devem ser providas. Assim, tomando o exemplo do código-fonte, é extremamente pertinente a definição de um padrão de codificação, indicando nomes de variáveis válidos, estilos de identação, regras para comentários etc.

Padrões organizacionais, sejam de processo sejam de produto, são muito importantes, pois eles fornecem um meio de capturar as melhores práticas de uma organização e facilitam a realização de atividades de avaliação da qualidade, que podem ser dirigidas pela avaliação da conformidade em relação ao padrão. Além disso, sendo organizacionais, todos os membros da organização tendem a estar familiarizados com os mesmos, facilitando a manutenção dos artefatos ou a substituição de pessoas no decorrer do projeto. Para aqueles artefatos tidos como os mais importantes no planejamento da documentação, é saudável que haja um padrão organizacional associado.

Dada a importância de padrões organizacionais, eles devem ser elaborados com bastante cuidado. Uma boa prática, conforme já sugerido para a definição de processos

padrão, consiste em usar como base padrões gerais propostos por instituições nacionais ou internacionais voltadas para a área de qualidade de software, tal como a ISO.

Revisões

Para se controlar a qualidade em um projeto de software, uma abordagem bastante usada consiste em se realizar revisões. Nas revisões, processos, documentos e outros artefatos são revisados por um grupo de pessoas, com o objetivo de verificar se os mesmos estão em conformidade com os padrões organizacionais estabelecidos e se o propósito de cada um deles está sendo atingido, incluindo o atendimento a requisitos do cliente e usuários. Assim, o objetivo de uma revisão é detectar erros e inconsistências em artefatos e processos, sejam eles relacionados à forma, sejam em relação ao conteúdo, e apontá-los aos responsáveis pela sua elaboração.

O processo de revisão começa com o planejamento da revisão, quando uma equipe de revisão é formada, tendo à frente um líder. A equipe de revisão deve incluir os membros da equipe que possam efetivamente úteis para atingir o objetivo da revisão. Muitas vezes, a pessoa responsável pela elaboração do artefato a ser revisado integra a equipe de revisão.

Dando início ao processo de revisão propriamente dito, normalmente, o autor do artefato apresenta o mesmo e descreve a perspectiva utilizada para a sua construção. Além disso, o propósito da revisão deve ser previamente informado e o material a ser revisado deve ser entregue com antecedência para que cada membro da equipe de revisão possa avaliá-lo. Uma vez que todos estejam preparados, uma reunião é convocada pelo líder. Essa reunião deverá ser relativamente breve (duas horas, no máximo), uma vez que todos já estão preparados para a mesma. Durante a reunião, o líder orientará o processo de revisão, passando por todos os aspectos relevantes a serem revistos. Todas as considerações dos demais membros da equipe de revisão devem ser discutidas e as decisões registradas, dando origem a uma ata de reunião de revisão, contendo uma lista de defeitos encontrados.

Gerência de Configuração

Referências: [1] Cap. 9; [2] Cap. 29; [3] Cap. 3, seção 3.3.

Durante o processo de desenvolvimento de software, vários artefatos são produzidos e alterados constantemente, evoluindo até que seus propósitos fundamentais sejam atendidos. Ferramentas de software, tais como compiladores e editores de texto, e processos também podem ser substituídos por versões mais recentes ou mesmo por outras, no caso de ferramentas. Porém, caso essas mudanças não sejam devidamente documentadas e comunicadas, poderão acarretar diversos problemas, tais como: dois ou mais desenvolvedores podem estar alterando um mesmo artefato ao mesmo tempo; não se saber qual a versão mais atual de um artefato; não se refletir alterações nos artefatos impactados por um artefato em alteração. Esses problemas podem gerar vários transtornos como incompatibilidade entre os grupos de desenvolvimento, inconsistências, retrabalho, atraso na entrega e insatisfação do cliente.

Assim, para que esses transtornos sejam evitados, é de suma importância o acompanhamento e o controle de artefatos, processos e ferramentas, através de um processo de gerência de configuração de software, durante todo o ciclo de vida do software [5].

A Gerência de Configuração de Software (GCS) visa estabelecer e manter a integridade dos itens de software ao longo de todo o ciclo de vida do software, garantindo a completeza, a consistência e a correção de tais itens, e controlando o armazenamento, a manipulação e a distribuição dos mesmos. Para tal, tem de identificar e documentar os produtos de trabalho que podem ser modificados, estabelecer as relações entre eles e os mecanismos para administrar suas diferentes versões, controlar modificações e permiter auditoria e a elaboração de relatórios sobre o estado de configuração.

Pelos objetivos da GCS, pode-se notar que ela está diretamente relacionada com as atividades de garantia da qualidade de software.

As atividades da GCS podem ser resumidas em:

- Planejamento da GCS: Um plano deve ser elaborado descrevendo as atividades da gerência de configuração, procedimentos e responsáveis pela execução dessas atividades.
- Identificação da Configuração: refere-se à identificação dos itens de software e suas versões a serem controladas, estabelecendo linhas básicas.
- Controle de Versão: combina procedimentos e ferramentas para administrar diferentes versões dos itens de configuração criados durante o processo de software.
- Controle de Modificação: combina procedimentos humanos e ferramentas automatizadas para controlar as alterações feitas em itens de software. Para tal, o seguinte processo é normalmente realizado: solicitação de mudança, aprovação ou rejeição da solicitação, registro de retirada para alteração (*check-out*), análise, avaliação e realização das alterações, revisão e registro da realização das alterações (*check-in*).
- Auditoria de Configuração: visa avaliar um item de configuração quanto a características não consideradas nas revisões, tal como se os itens relacionados aos solicitados foram devidamente atualizados.
- Relato da situação da configuração: refere-se à preparação de relatórios que mostrem a situação e o histórico dos itens de software controlados. Tais relatórios podem incluir, dentre outros, o número de alterações nos itens, as últimas versões dos mesmos e identificadores de liberação.

O Processo de GCS

O primeiro passo do processo de gerência de configuração de software é a confecção de um plano de gerência de configuração, que inicia com a identificação dos itens que serão colocados sob gerência de configuração, chamados itens de configuração. Os itens mais relevantes para serem submetidos à gerência de configuração são aqueles mais usados durante o ciclo de vida, os mais genéricos, os mais importantes para segurança, os projetados para reutilização e os que podem ser modificados por vários desenvolvedores ao mesmo tempo [6]. Os itens não colocados sob gerência de configuração podem ser alterados livremente.

Após a seleção dos itens, deve-se descrever como eles se relacionam. Isso será muito importante para as futuras manutenções, pois permite identificar de maneira eficaz os itens afetados em decorrência de uma alteração. Além disso, deve-se criar um esquema de

identificação dos itens de configuração, com atribuição de nomes exclusivos, para que seja possível estabelecer a evolução de cada versão dos itens [1, 6].

Após a identificação dos itens de configuração, devem ser planejadas as linhas-base dentro do ciclo de vida do projeto. Uma linha-base (ou *baseline*) é uma versão estável de um sistema contendo todos os componentes que constituem este sistema em um determinado momento. Nos pontos estabelecidos pelas linhas-base, os itens de configuração devem ser identificados, analisados, corrigidos, aprovados e armazenados em um local sob controle de acesso, denominado repositório central, base de dados de projeto ou biblioteca de projeto. Assim, quaisquer alterações nos itens daí em diante só poderão ser realizadas através de procedimentos formais de controle de modificação [1, 6].

O passo seguinte do processo de GCS é o controle de versão, que combina procedimentos e ferramentas para identificar, armazenar e administrar diferentes versões dos itens de configuração que são criados durante o processo de software [1, 6]. A idéia é que a cada modificação que ocorra em um item de configuração, uma nova versão ou variante seja criada. Versões de um item são geradas pelas diversas alterações, enquanto variantes são as diferentes formas de um item, que existem simultaneamente, e atendem a requisitos similares [6].

Talvez a mais importante atividade do processo de GCS é o controle de alterações, que combina procedimentos humanos e ferramentas automatizadas para o controle das alterações realizadas nos itens de configuração [1]. Assim que uma alteração é solicitada, o impacto em outros itens de configuração e o custo para a modificação devem ser avaliados. Um responsável deve decidir se a alteração poderá ou não ser realizada. Caso a alteração seja liberada, pessoas são indicadas para sua execução. Assim que não houver ninguém utilizando os itens de configuração envolvidos, cópias deles são retiradas do repositório central e colocadas em uma área de trabalho do desenvolvedor, através de um procedimento denominado *check-out*. A partir deste momento, nenhum outro desenvolvedor poderá alterar esses itens. Os desenvolvedores designados fazem as alterações necessárias e, assim que essas forem concluídas, os itens são submetidos a uma revisão. Se as alterações forem aprovadas, os itens são devolvidos ao repositório central, estabelecendo uma nova linha-base, em um procedimento chamado *check-in* [1, 6].

Porém, mesmo com os mecanismos de controle mais bem sucedidos, não é possível garantir que as modificações foram corretamente implementadas. Assim, revisões e auditorias de configuração de software são necessárias no processo de GCS [1]. Essas atividades de garantia da qualidade tentam descobrir omissões ou erros na configuração e se os procedimentos, padrões, regulamentações ou guias foram devidamente aplicados no processo e no produto [6].

Enfim, o último passo do processo de GCS é a preparação de relatórios, que é uma tarefa que tem como objetivo relatar a todas as pessoas envolvidas no desenvolvimento e manutenção do software as seguintes questões: (i) O que aconteceu? (ii) Quem fez? (iii) Quando aconteceu? (iv) O que mais será afetado? O acesso rápido às informações agiliza o processo de desenvolvimento e melhora a comunicação entre as pessoas, evitando, assim, muitos problemas de alterações do mesmo item de configuração, com intenções diferentes e, às vezes, conflitantes [6].

Métricas e Medição

Referências: [1] Cap. 4, seções 4.1, 4.2 e 4.3; [2] Cap. 24, seção 24.4;

Para poder controlar a qualidade, medir é muito importante. Se não é possível medir, expressando em números, apenas uma análise qualitativa (e, portanto, subjetiva) pode ser feita, o que, na maioria das vezes, é insuficiente. Com medições, as tendências (boas ou más) podem ser detectadas, melhores estimativas podem ser feitas e melhorias reais podem ser conseguidas [1].

Quando se trata de avaliação quantitativa, quatro conceitos, muitas vezes usados equivocadamente com o mesmo sentido, são importantes: medida, medição, métrica e indicador. Uma **medida** fornece uma indicação quantitativa da extensão, quantidade, dimensão, capacidade ou tamanho de um atributo de um produto ou de um processo. Quando os dados de um único ponto são coletados, uma medida é estabelecida (p.ex., quantidade de erros descobertos em uma revisão). **Medição** é o ato de medir, isto é, de determinar uma medida. Já uma **métrica** procura relacionar medidas individuais com o objetivo de se ter uma idéia da eficácia do processo, do projeto ou do produto sendo medido. Por fim, desenvolve-se métricas para se obter **indicadores**, isto é, para se ter uma compreensão do processo, produto ou projeto sendo medido.

Seja o seguinte exemplo: deseja-se saber se uma pessoa está com seu peso ideal ou não. Para tal, duas **medidas** são importantes: altura (H) e peso (P). Ao medir essas dimensões, está-se efetuando uma **medição**. A **métrica** "índice de massa corporal (IMC)" é calculada segundo a seguinte fórmula: $IMC = P / H^2$. A partir dessa métrica, a Organização Mundial de Saúde estabeleceu **indicadores** que apontam se um adulto está acima do peso, se está obeso ou abaixo do peso ideal considerado saudável, conforme abaixo:

Se IMC < 18,5, então o indivíduo está abaixo do peso ideal considerado saudável;

Se 18,5 <= IMC < 25, então o indivíduo está com o peso normal;

Se 25 <= IMC <= 30, então o indivíduo está acima do peso;

Se IMC > 30, então o indivíduo está obeso.

Realizando medições, uma pessoa pode obter suas medidas para altura e peso. A partir delas, pode calcular a métrica IMC e ter um indicador se está abaixo do peso, no peso ideal, acima do peso ou obeso. Conhecendo esse indicador, a pessoa pode ajustar seu processo de alimentação, obtendo melhorias reais para sua saúde.

Uma vez que a medição de software se preocupa em obter valores numéricos (medidas) para alguns atributos de um produto ou de um processo, uma questão importante passa a ser: Que atributos medir?

O modelo de qualidade definido na norma ISO/IEC 9126-1 trata dessa questão, estando sub-dividido em dois modelos: o modelo de qualidade para características externas e internas e o modelo de qualidade para qualidade em uso. O modelo de qualidade para características externas e internas classifica os atributos de qualidade de software em seis características (funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade) que são, por sua vez, desdobradas em sub-características. As sub-características podem ser desdobradas em mais níveis, até se ter sub-características diretamente mensuráveis, para as quais métricas são aplicadas. As normas ISO/IEC 9126-2 e 9126-3 apresentam, respectivamente, métricas externas e internas.

Esse modelo de qualidade que preconiza a análise de características de qualidade a partir de suas sub-características de forma recursiva até que se tenham métricas para coletar dados é aplicável não somente a produto, mas também para avaliar a qualidade de processos de software. Assim, de maneira geral, na avaliação quantitativa da qualidade, é necessário:

1. Definir características de qualidade relevantes para avaliar a qualidade do objeto em questão (produto ou processo).
2. Para cada característica de qualidade selecionada, definir sub-características de qualidade relevantes que tenham influência sobre a mesma, estabelecendo um modelo ou fórmula de computar a característica a partir das sub-características. Fórmulas baseadas em peso são bastante utilizadas: $cq = p_1 * scq_1 + ... + p_n * scq_n$.
3. Usar procedimento análogo ao anterior para as sub-características não passíveis de mensuração direta.
4. Para as sub-características diretamente mensuráveis, selecionar métricas, coletar medidas e computar as métricas segundo a fórmula ou modelo estabelecido.
5. Fazer o caminho inverso, agora computando sub-características não diretamente mensuráveis, até se chegar a um valor para a característica de qualidade.

Concluído o processo de medição, deve-se comparar os valores obtidos com padrões estabelecidos para a organização, de modo a se obter os indicadores da qualidade. A partir dos indicadores, ações devem ser tomadas visando à melhoria.

Vale destacar que, especialmente no caso da avaliação da qualidade de software, métricas relacionadas a defeitos são bastante úteis, tal como (número de erros / ponto de função).

O único modo racional de melhorar um processo é medir atributos específicos, obter um conjunto de métricas significativas baseadas nesses atributos e usar os valores das métricas para fornecer indicadores que conduzirão um processo de melhoria [1].

Referências

1. R.S. Pressman, *Engenharia de Software*, Rio de Janeiro: McGraw Hill, 5ª edição, 2002.
2. I. Sommerville, *Engenharia de Software*, São Paulo: Addison-Wesley, 6ª edição, 2003.
3. S.L. Pfleeger, *Engenharia de Software: Teoria e Prática*, São Paulo: Prentice Hall, 2ª edição, 2004.
4. R. Sanches, "Documentação de Software". In: *Qualidade de Software: Teoria e Prática*, Eds. A.R.C. Rocha, J.C. Maldonado, K. Weber, Prentice Hall, 2001.
5. J.C. Maldonado, S.C.P.F. Fabbri, "Verificação e Validação de Software". In: *Qualidade de Software: Teoria e Prática*, Eds. A.R.C. Rocha, J.C. Maldonado, K. Weber, Prentice Hall, 2001.
6. R. Sanches, "Gerência de Configuração de Software". In: *Qualidade de Software: Teoria e Prática*, Eds. A.R.C. Rocha, J.C. Maldonado, K. Weber, Prentice Hall, 2001.

5 – Levantamento e Análise de Requisitos

Referências: [1] Cap. 11, [2] Cap. 5, [3] Cap. 4.

Uma das principais medidas do sucesso de um software é o grau no qual ele atende aos objetivos e requisitos para os quais foi construído. De forma geral, a Engenharia de Requisitos de Software é o processo de identificar todos os envolvidos, descobrir seus objetivos e necessidades e documentá-los de forma apropriada para análise, comunicação e posterior implementação [4].

Requisito e Tipos de Requisitos

As descrições das funções que um sistema deve incorporar e das restrições que devem ser satisfeitas são os requisitos para o sistema. Isto é, os requisitos de um sistema definem o que o sistema deve fazer e as circunstâncias sob as quais deve operar. Em outras palavras, os requisitos definem os serviços que o sistema deve fornecer e dispõem sobre as restrições à operação do mesmo [2].

Requisitos são, normalmente, classificados em requisitos funcionais e não funcionais. Requisitos funcionais, como o próprio nome indica, apontam as funções que o sistema deve fornecer e como o sistema deve se comportar em determinadas situações. Já os requisitos não funcionais descrevem restrições sobre as funções oferecidas, tais como restrições de tempo, de uso de recursos etc. Alguns requisitos não funcionais dizem respeito ao sistema como um todo e não a funcionalidade específica. Dependendo da natureza, os requisitos não funcionais podem ser classificados de diferentes maneiras, tais como requisitos de desempenho, requisitos de portabilidade, requisitos legais, requisitos de conformidade etc [2].

O Processo da Engenharia de Requisitos

O processo de descobrir, analisar, documentar e verificar / validar requisitos é chamado de processo de engenharia de requisitos [2]. Os processos de engenharia de requisitos variam muito de uma organização para outra, mas, de maneira geral, a maioria dos processos de Engenharia de Requisitos é composta das seguintes atividades [4]:

- Elicitação / Descoberta / Levantamento de Requisitos: Nesta fase, os usuários, clientes e especialistas de domínio são identificados e trabalham junto com os engenheiros de requisitos para descobrir, articular e entender a organização como um todo, o domínio da aplicação, os processos de negócio específicos, as necessidades que o software deve atender e os problemas e deficiências dos sistemas atuais. Os diferentes pontos de vista dos participantes do processo, bem como as oportunidades e restrições existentes, os problemas a serem resolvidos, o desempenho requerido e as restrições também devem ser levantados.
- Análise de Requisitos: visa a estabelecer um conjunto acordado de requisitos consistentes e sem ambigüidades, que possa ser usado como base para o desenvolvimento do software. Para tal, diversos tipos de modelos são construídos. Geralmente, a análise de requisitos inclui também a negociação para resolver conflitos detectados.

- Documentação de Requisitos: é a atividade de representar os resultados da Engenharia de Requisitos em um documento, contendo os requisitos do software.
- Verificação e Validação de Requisitos: A verificação de requisitos avalia se o documento de Especificação de Requisitos está sendo construído de forma correta, de acordo com padrões previamente definidos, sem conter requisitos ambíguos, incompletos ou, ainda, requisitos incoerentes ou impossíveis de serem testados. Já a validação diz respeito a avaliar se esse documento está correto, ou seja, se os requisitos e modelos documentados atendem às reais necessidades e requisitos dos usuários / cliente.
- Gerência de Requisitos: se preocupa em gerenciar as mudanças nos requisitos já acordados, manter uma trilha dessas mudanças, gerenciar os relacionamentos entre os requisitos e as dependências entre o documento de requisitos e os demais artefatos produzidos no processo de software, de forma a garantir que mudanças nos requisitos sejam feitas de maneira controlada e documentada.

É possível notar que, das cinco atividades do processo de Engenharia de Requisitos anteriormente listadas, as três últimas são, na realidade, instanciações para a Engenharia de Requisitos de atividades genéricas já discutidas no capítulo 4, a saber: documentação, garantia da qualidade e gerência de configuração. Assim sendo, neste capítulo, somente as duas primeiras atividades (Levantamento e Análise de Requisitos) são discutidas.

Levantamento de Requisitos

O levantamento de requisitos é uma atividade complexa que não se resume somente a perguntar às pessoas o que elas desejam e também não é uma simples transferência de conhecimento. Várias técnicas de levantamento de requisitos são normalmente empregadas pelos engenheiros de requisitos (ou analistas de sistemas), dentre elas entrevistas, questionários, prototipação, investigação de documentos, observação, dinâmicas de grupo etc.

Análise de Requisitos

A análise de requisitos (muitas vezes chamada análise de sistemas) é, em última instância, uma atividade de construção de modelos. Um modelo é uma representação de alguma coisa do mundo real, uma abstração da realidade, e, portanto, representa uma seleção de características do mundo real relevantes para o propósito do sistema em questão.

Modelos são fundamentais no desenvolvimento de sistemas. Tipicamente eles são construídos para:

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

Um modelo enfatiza um conjunto de características da realidade, que corresponde à *dimensão do modelo*. Além da dimensão que um modelo enfatiza, modelos possuem níveis de abstração. O *nível de abstração* de um modelo diz respeito ao grau de detalhamento com que

as características do sistema são representadas. Em cada nível há uma ênfase seletiva nos detalhes representados. No caso do desenvolvimento de sistemas, geralmente, são considerados três níveis:

- *conceitual*: considera características do sistema independentes do ambiente computacional (hardware e software) no qual o sistema será implementado. Essas características são dependentes unicamente das necessidades do usuário. Modelos conceituais são construídos na atividade de análise de requisitos.
- *lógico*: características dependentes de um determinado *tipo* de sistema computacional. Essas características são, contudo, independentes de produtos específicos. Tais modelos são típicos da fase de projeto.
- *físico*: características dependentes de um sistema computacional específico, isto é, uma linguagem e um compilador específico, um sistema gerenciador de bancos de dados específico, o hardware de um determinado fabricante etc. Tais modelos podem ser construídos tanto na fase de projeto quanto na fase de implementação.

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

Para a realização da atividade de análise, uma escolha deve ser feita: o paradigma de desenvolvimento. Paradigmas de desenvolvimento estabelecem a forma de se ver o mundo e, portanto, definem as características básicas dos modelos a serem construídos. Por exemplo, o paradigma orientado a objetos parte do pressuposto que o mundo é povoado por objetos, ou seja, a abstração básica para se representar as coisas do mundo são os objetos. Já o paradigma estruturado adota uma visão de desenvolvimento baseada em um modelo entrada-processamento-saída. No paradigma estruturado, os dados são considerados separadamente das funções que os transformam e a decomposição funcional é usada intensamente.

Neste texto, as atividades de Levantamento e Análise de Requisitos são discutidas à luz do paradigma estruturado, mais especificamente utilizando os conceitos da Análise Essencial de Sistemas.

5.1 – Análise Essencial de Sistemas

Referência Básica: [5]

O método da Análise Essencial de Sistemas [5] preconiza que, de uma forma geral, um sistema deve ser modelado através de três dimensões:

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

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

- *Modelo Essencial*: representa o sistema num grau de abstração completamente independente de restrições tecnológicas.

- *Modelo de Implementação*: passa a considerar as restrições tecnológicas impostas pela plataforma de hardware e software a ser utilizada para implementar o sistema.

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

A Análise Essencial é, de fato, uma evolução da Análise Estruturada de Sistemas, mais especificamente da Análise Estruturada Moderna [6]. Os conceitos introduzidos pela Análise Essencial endereçavam inicialmente as duas principais dificuldades que os analistas enfrentavam com a aplicação da Análise Estruturada: a distinção entre requisitos lógicos e físicos, e a ausência de uma abordagem para dividir o sistema em partes tão independentes quanto possível, de modo a facilitar o processo de análise.

Durante muito tempo, no paradigma estruturado, houve grandes debates entre os profissionais de desenvolvimento de sistemas sobre por qual perspectiva se deveria começar a especificação de um sistema: pelos dados ou pelas funções? Os argumentos, igualmente válidos, exploravam considerações como:

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

A Análise Essencial procurou estabelecer um novo ponto de partida para a especificação de um sistema: a identificação dos *eventos* que o afetam [5].

Conforme apontado anteriormente, um problema bastante relevante na especificação consiste em decidir como efetuar uma boa divisão do problema. A Análise Estruturada propõe que essa divisão seja obtida por meio de uma abordagem *top-down*. Embora essa seja uma boa maneira de se atacar um problema complexo – partir da visão geral e ir descendo, em uma visão hierárquica, a níveis de detalhes cada vez maiores – na prática, essa abordagem não se mostrou eficiente como estratégia para a decomposição de sistemas. A Análise Essencial propõe uma outra forma de particionamento, a qual é baseada nos eventos, e que mostrou ser mais efetiva do que a abordagem *top-down*, pois torna mais fácil a identificação das funções e entidades que compõem o sistema [5].

A Análise Essencial de Sistemas, através da técnica de particionamento por eventos, oferece uma boa estratégia para modelar o comportamento do sistema, visando satisfazer os requisitos do usuário, pressupondo-se que dispomos de tecnologia perfeita e que ela pode ser obtida a custo zero [7].

A Análise Essencial preserva todos os modelos da Análise Estruturada. De fato, embora diferentes, a melhor maneira de encarar a Análise Essencial é considerá-la uma evolução da Análise Estruturada. Isso porque a Análise Essencial introduz novos conceitos e abordagens, dentre eles: tecnologia perfeita, requisitos verdadeiros e falsos, eventos e respostas, atividades essenciais e memória essencial, conforme discutido a seguir.

Tecnologia Perfeita

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

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

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

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

Considerando que a tecnologia a ser utilizada na implementação do sistema é perfeita, todas as perguntas anteriores deixam de ter importância e, portanto, não precisam estar no foco do analista. Assim sendo, para distinguir um requisito lógico de um requisito físico, utilizando a abstração de tecnologia perfeita, formule a seguinte pergunta ao identificar um requisito qualquer: "Para atender ao seu propósito, o sistema necessitará possuir essa capacidade ou essa característica, mesmo considerando que ele será implementado em uma tecnologia perfeita?" Se a resposta for sim, esse requisito é verdadeiro e deve ser modelado.

Requisito Verdadeiro e Requisito Falso

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

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

Evento, Estímulo, Ação e Resposta

Evento e resposta são nomes genéricos de interações entre o ambiente externo e o sistema. Um *evento* pode ser definido informalmente como um acontecimento do mundo exterior que requer do sistema uma *resposta*. Corresponde a alguma mudança no ambiente externo que funcionará como um *estímulo* para o sistema, isto é, o sistema deve responder a esse estímulo para atender ao seu propósito. Uma *resposta* é o resultado da execução de um conjunto de *ações* no sistema, como conseqüência do reconhecimento pelo sistema de que um evento ocorreu. Uma resposta tipicamente pode ser [5]:

- um fluxo de dados saindo do sistema para uma entidade externa;

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

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

Os eventos são classificados em três tipos diferentes, dependendo da maneira como ocorrem e da natureza do estímulo que produzem [5]:

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

  <*Entidade externa que provocou o evento*> +

  <*ação* – verbo na voz ativa> +

  <*estímulo* – fluxo de dados enviado ao sistema>

  Ex.: Cliente envia pedido.

  Cliente cancela pedido.

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

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

    <*Entidade externa que provocou o evento*> +

    <*ação* – verbo na voz ativa> +

    <*complemento*>

    Ex.: Gerente solicita relação de clientes.

    Diretoria autoriza o pagamento de uma fatura.

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

    <*Sujeito*> + <*ação* – verbo na voz passiva>

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

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

sistema toma conhecimento da ocorrência do evento. Os eventos temporais podem ser nomeados da seguinte forma:

<Indicação de tempo> + <ação> + <complemento>

Ex.: Mensalmente, emitir relatório de vendas.

Atividades Essenciais

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

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

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

Memória Essencial

A memória essencial corresponde ao conjunto mínimo de dados que deve ser armazenado pelo sistema para atender ao seu propósito, considerando que ele será implementado em uma tecnologia perfeita.

O modelo normalmente utilizado para modelar a memória essencial é o Modelo de Entidades e Relacionamentos (MER). Nos Diagramas de Fluxo de Dados (DFDs), a memória essencial também é representada, neste caso pelos depósitos de dados. Assim, ao se considerar a memória essencial de um sistema, essas duas visões da mesma têm de estar consistentes. Para garantir essa consistência a seguinte regra deve ser observada: cada depósito de dados de um DFD deve corresponder a uma entidade ou a um relacionamento com atributos do MER.

Para levar em conta a abstração da tecnologia perfeita, os depósitos de dados / entidades não devem considerar a forma como os relacionamentos entre eles são estabelecidos. Por exemplo, quando se utilizam bancos de dados relacionais, chaves estrangeiras (atributos determinantes transpostos entre entidades) são armazenadas para representar um relacionamento entre entidades. Entretanto, deve-se ressaltar que essa é uma característica específica dos bancos de dados relacionais, uma tecnologia nada perfeita. Lembre-se que, na fase de análise, a tecnologia de implementação ainda não foi selecionada e deve ser considerada perfeita.

5.2 - Especificação da Essência do Sistema

Referência: [5] Capítulo 17

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

A especificação da essência do sistema, produto das fases de levantamento e análise de requisitos, é composta de dois modelos, como mostra a figura 5.1:

- Modelo Ambiental: define a fronteira entre o sistema e o resto do mundo. Oferece uma perspectiva externa do sistema e é o principal produto da atividade de levantamento de requisitos.
- Modelo Comportamental: define o comportamento do sistema necessário para interagir com o ambiente. Explora características internas do sistema e é o principal produto da atividade de análise de requisitos.

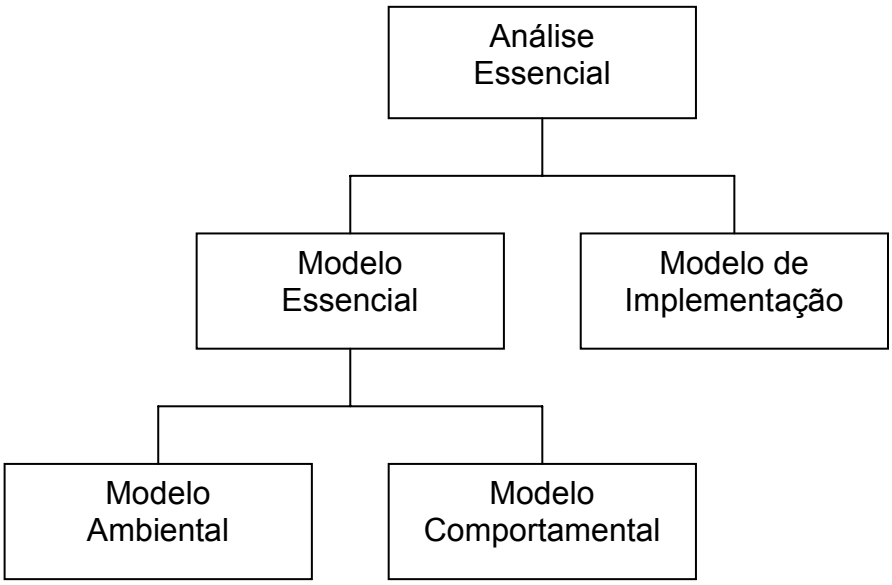


Figura 5.1 – A Análise Essencial e seus Modelos.

5.2.1 - O Modelo Ambiental

Referência: [5] Seção 17.1

Representa o que o sistema deve fazer para atender ao ambiente, segundo uma perspectiva externa, ou seja, do usuário. É composto dos seguintes artefatos:

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

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

A lista de eventos é o produto principal do levantamento de requisitos quando a Análise Essencial é adotada, uma vez que os eventos constituem uma parte fundamental da abordagem desse método. De fato, o primeiro passo na especificação de um sistema é identificar a quais eventos do mundo exterior ele deverá responder.

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

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

Lista de Eventos

Referência: [5] Capítulo 15

Dado que a Lista de Eventos é o principal produto do levantamento de requisitos segundo a Análise Essencial, vale um exame mais cuidadoso desse artefato. Um formato largamente utilizado é o mostrado na tabela 5.1, que apresenta parte de uma lista de eventos para um sistema de biblioteca.

Evento	Tipo	Estímulo	Ação	Resposta
Bibliotecário cadastra dados de usuário	FD	dados-usuário	Cadastrar Usuário	(usuário cadastrado)
Diariamente, cancelar reservas vencidas	T	-	Cancelar Reservas	(reservas vencidas excluídas)
Usuário consulta livros	C	solicitação de consulta, parâmetro da consulta (título, autor ou assunto)	Consultar Livro	dados-livros

Tabela 5.1 – Exemplo de uma lista de eventos.

Nas duas primeiras colunas, o evento a ser tratado é descrito (segundo as regras de nomeação de eventos anteriormente apresentadas) e classificado – evento orientado por fluxo de dados (FD), orientado por controle (C) e evento temporal (T).

Na terceira coluna, são listados os estímulos que apontam para o sistema a ocorrência do evento. No caso dos eventos orientados por fluxo de dados, tipicamente um fluxo de dados representa o estímulo. Para os eventos temporais, é a passagem do tempo, considerado um estímulo implícito e, portanto, não é necessário representá-lo. Para os eventos orientados por controle, um fluxo de controle informando a solicitação feita ao sistema é o principal estímulo. Vale lembrar que, tanto no caso dos eventos temporais quanto dos eventos orientados a controle, fluxos de dados auxiliares podem ser também parte do estímulo. Neste caso, eles também devem aparecer.

Por fim, as duas últimas colunas registram a ação a ser feita pelo sistema e a resposta produzida. Quando a resposta for interna ao sistema, ela é escrita entre parênteses.

Para sistemas de médio a grande porte, a lista de eventos pode ser muito extensa. Um bom recurso para tratar a complexidade neste caso, pode ser a construção de listas de eventos separadas para cada um dos subsistemas identificados. Além disso, atividades custodiais que apenas cadastram dados no sistema, abrangendo a inclusão de um novo item, alteração de dados de um item existente, consulta básica ao item e exclusão de um item, podem ser tratadas como um único evento "Cadastrar Item", classificado como orientado a fluxo de dados, ainda que a consulta e a exclusão sejam tipicamente eventos orientados por controle. A premissa para essa simplificação é que, na maioria dos sistemas, haverá um grande número de eventos desta natureza e que seguem um comportamento bastante padrão e conhecido, não sendo necessário detalhá-lo.

5.2.2 – O Modelo Comportamental

Referência: [5] Seção 17.2

O modelo comportamental representa o que o interior do sistema deve fazer para atender aos eventos apontados pelo modelo ambiental, sendo o principal produto da análise de requisitos segundo o método da Análise Essencial. Esse modelo contém os seguintes artefatos:

- **Diagrama de Entidades e Relacionamentos (DER)**
- **Diagramas de Fluxos de Dados (DFDs) Particionados por Eventos**: Para cada evento do sistema, um DFD pode ser elaborado. Assim, a quantidade de diagramas pode chegar ao número de eventos na lista.
- **Diagramas de Estados**: Representa o comportamento das entidades e relacionamentos com atributos ao longo do tempo. Tipicamente, constrói-se um diagrama de estados para cada entidade ou relacionamento com atributo do DER que possuir comportamento significativo, isto é, possuir mais de um estado ao longo de seu ciclo de vida.
- **Diagramas de Fluxos de Dados Organizados em Níveis Hierárquicos**: representa os processos em níveis hierárquicos, a partir do diagrama zero. Os processos do diagrama zero são obtidos através do agrupamento de atividades essenciais dos DFDs particionados por eventos. Um critério de agrupamento bastante razoável é considerar o grau de coesão e acoplamento entre atividades essenciais. As seguintes heurísticas podem ser utilizadas, em conjunto ou em separado:
  - Procurar agrupar em um único processo todas as atividades essenciais que acessam um determinado depósito de dados, verificando se o processo resultante desse agrupamento é adequado para representar uma das funções do sistema.
  - Agrupar todas as atividades de custódia referentes a um mesmo depósito de dados.
  - Procurar identificar uma função do sistema, agrupando atividades essenciais que interagem com uma mesma entidade externa.

- Representar no DFD-zero, um processo para cada uma das funções do negócio. Agrupar as atividades essenciais nos processos para os quais as suas ações mais contribuem.

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

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

Como se pode perceber, a Análise Essencial faz uso praticamente das mesmas técnicas de modelagem da Análise Estruturada, a saber a Modelagem de Dados (utilizando modelos de Entidades e Relacionamentos), a Modelagem Funcional (utilizando Diagramas de Fluxo de Dados – DFDs) e a Modelagem de Controle (utilizando Diagramas de Estados). Isso é bastante natural, já que a Análise Essencial é, de fato, uma extensão da Análise Estruturada.

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

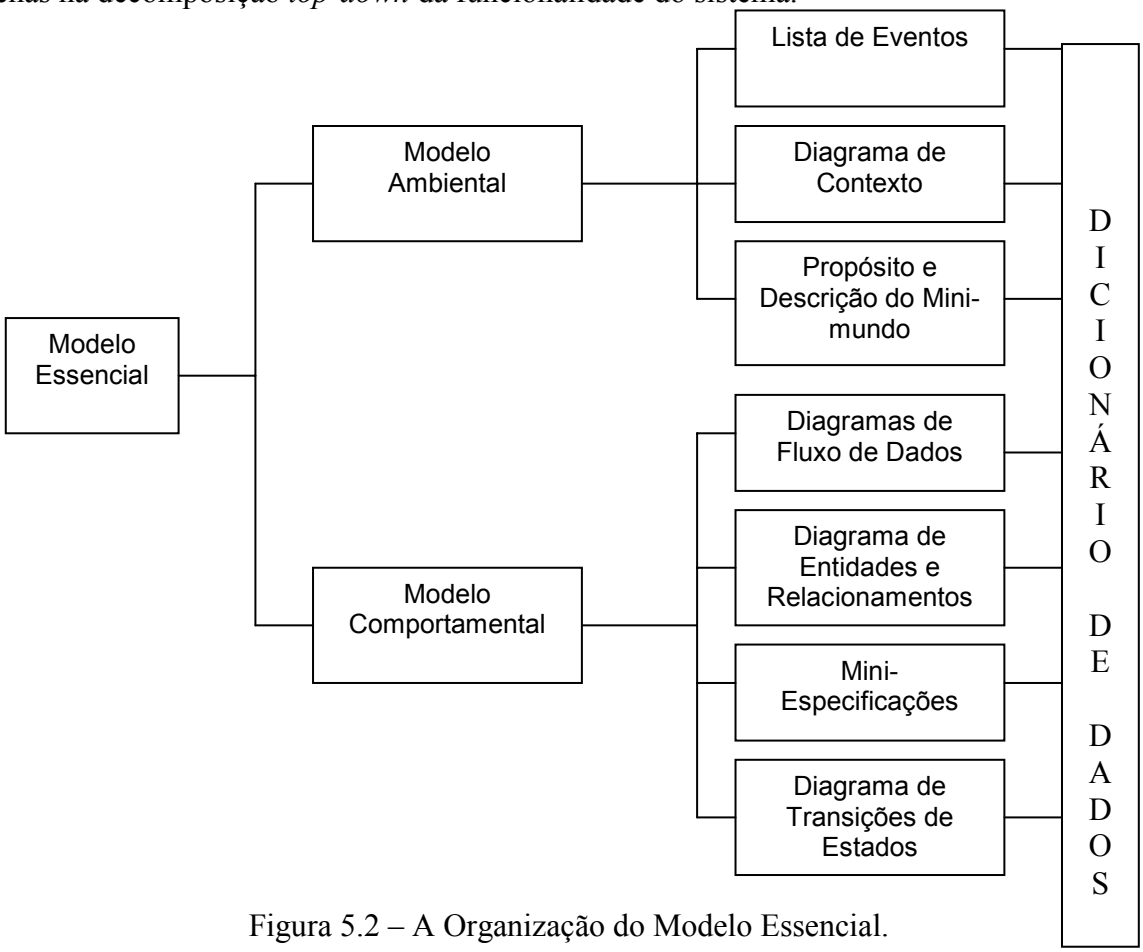


Figura 5.2 – A Organização do Modelo Essencial.

5.3. Modelagem de Dados

A primeira atividade realizada no processo de construção do Modelo Comportamental da Análise Essencial de Sistemas deve ser a modelagem de dados e, para tal, o modelo de Entidades e Relacionamentos (ER) é utilizado. O modelo ER é uma técnica utilizada para representar os dados a serem armazenados em um sistema, tendo sido desenvolvida originalmente para dar suporte ao projeto de bancos de dados [8] [9].

5.3.1 – Diagrama de Entidades e Relacionamentos

Basicamente, um diagrama ER representa as *entidades* do mundo real e os *relacionamentos* entre elas, que um sistema de informação precisa simular internamente. Além disso, entidades e relacionamentos podem ter atributos.

Entidades

Uma **entidade** é uma representação abstrata de alguma coisa do mundo real que temos interesse em monitorar o comportamento. Pode ser a representação de um ser, um objeto, um organismo social etc. Assim, o funcionário João é uma entidade.

Entretanto, desejamos representar, de fato, **conjuntos de entidades**, isto é, grupos de entidades que têm características semelhantes. No modelo ER, conjuntos de entidades são representados graficamente por retângulos, como mostra a figura 5.3.

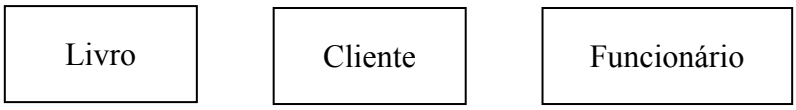


Figura 5.3 – Representação Gráfica de Conjuntos de Entidades.

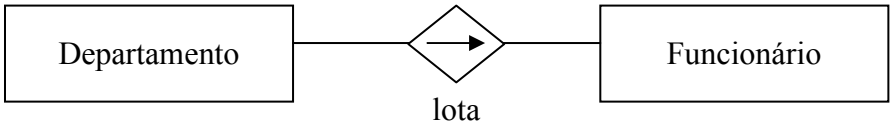
Um conjunto de entidades representa todos os elementos do mundo real referidos pelo conjunto. Por exemplo, em um sistema de uma biblioteca, o conjunto de entidades Livro representa todos os livros de uma biblioteca.

Para estabelecermos uma padronização, neste texto, usaremos nomes de conjuntos de entidades sempre no singular e escritos com a primeira letra maiúscula. No entanto, isto não representa efetivamente uma regra. Além disso, usaremos o termo entidade para referenciar tanto entidades quanto conjuntos de entidades, de maneira indistinta, ainda que sejam conceitos diferentes. Essa é uma prática bastante comum e o contexto será suficiente para que o leitor perceba se estamos tratando de um conjunto de entidades ou de uma entidade específica.

Relacionamentos

Um **relacionamento** é uma abstração de uma associação entre duas ou mais entidades. Por exemplo, podemos querer registrar que o funcionário João (uma entidade do conjunto Funcionário) está lotado (um relacionamento) no departamento de Vendas (uma entidade do conjunto Departamento). Um relacionamento Binário, como o citado no exemplo anterior, é uma representação abstrata da associação entre duas entidades.

Da mesma forma que com as entidades, estamos mais interessados em modelar **conjunto de relacionamentos**. Um conjunto de relacionamentos é um subconjunto do produto cartesiano dos conjuntos de entidades envolvidos, sendo representado por um losango com um verbo para indicar a ação e uma seta para informar o sentido de leitura, como mostra a figura 5.4. Além disso, assim como fizemos com entidades, usaremos indistintamente o termo relacionamento para designar tanto relacionamentos entre entidades específicas como para referenciar o conjunto de relacionamentos que existem entre conjuntos de entidades. Opcionalmente, usaremos o termo instância (tanto de entidades quanto de relacionamentos) para referenciar um elemento do conjunto (de entidades e de relacionamentos, respectivamente).



$lota \subseteq \{(d,f) / d \in$ Departamento e $f \in$ Funcionário$\}$

Figura 5.4 – Representação gráfica para relacionamentos.

É importante notar que todos os relacionamentos binários possuem uma leitura inversa. Se um departamento lota funcionários, então funcionários estão lotados em departamentos.

Conforme anteriormente mencionado, um conjunto de relacionamentos é um subconjunto do produto cartesiano das entidades envolvidas. É necessário, portanto, descrever de forma mais apurada qual é esse subconjunto. Isto é feito via definição de cardinalidades. Uma **cardinalidade** indica os números mínimo (cardinalidade mínima) e máximo (cardinalidade máxima) de associações possíveis em um relacionamento. Seja o seguinte caso: Um professor tem que estar lotado em um e somente um departamento, enquanto um departamento deve ter no mínimo 13 professores e no máximo um número arbitrário (N). Essa restrição imposta pelo mundo real deve ser considerada no modelo ER e ela é registrada usando-se cardinalidades, como mostra a figura 5.5.

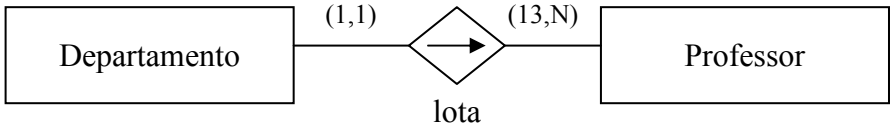


Figura 5.5 – Representação de cardinalidades em relacionamentos.

Vale destacar que a cardinalidade mínima aponta a quantidade de instâncias mínima necessária para que a associação seja estabelecida, considerando o momento em que uma instância de uma entidade é criada. Assim, no exemplo anterior, quando um novo professor for ser registrado no sistema, ele terá obrigatoriamente de estar lotado em um departamento.

Relacionamentos podem ser classificados em relação à cardinalidade mínima em relacionamentos totais e parciais. Dizemos que R é um relacionamento total em A se e somente se: $\forall\, a \in A, \exists\, b \in$ B / $(a, b) \in R$, isto é, todo elemento de A tem de participar obrigatoriamente de R e, conseqüentemente, a cardinalidade mínima de R em relação a A é 1. Por outro lado, dizemos que R é um relacionamento parcial em A se e somente se: $\exists\, a \in A$ / $\forall\, b \in$ B / $(a, b) \notin R$, isto é, nem todo elemento de A participa de R e, conseqüentemente, a cardinalidade mínima de R em relação a A é 0.

É importante frisar que, entre duas entidades, podem existir vários relacionamento diferentes, como mostra o exemplo da figura 5.6.

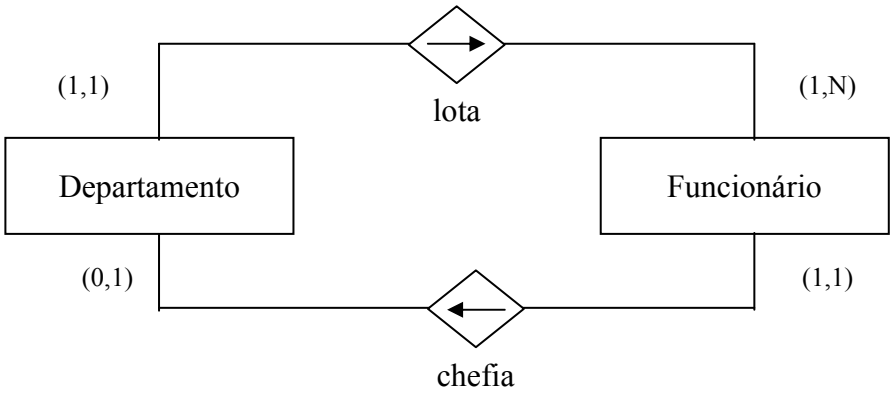


Figura 5.6 – Dois relacionamentos diferentes entre as mesmas entidades.

Além disso, uma entidade pode participar de relacionamentos com quaisquer outras entidades do modelo, inclusive com ela mesma (auto-relacionamento), como mostra a figura 5.7.

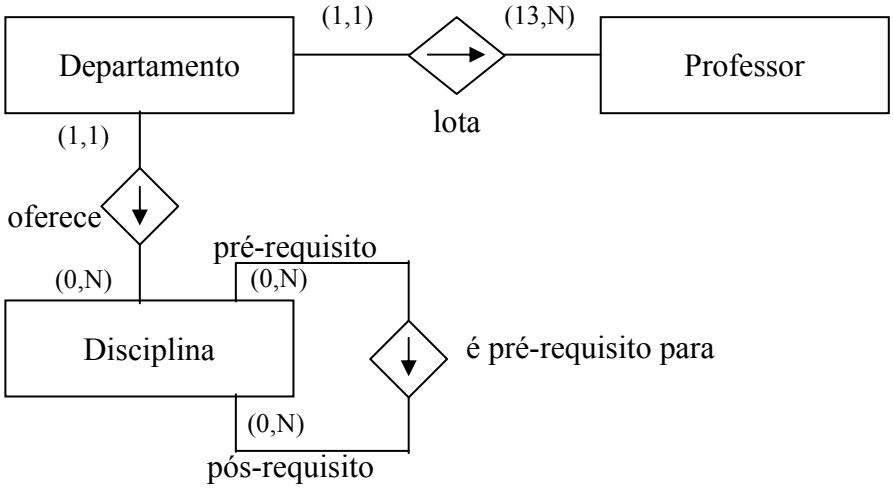


Figura 5.7 – Exemplo de auto-relacionamento.

No caso de auto-relacionamentos, é útil distinguir qual a atuação de cada elemento do conjunto de entidades no relacionamento e, portanto, é importante atribuir *papéis*. Assim no caso do auto-relacionamento *é pré-requisito para* da figura 5.7, estamos dizendo que:

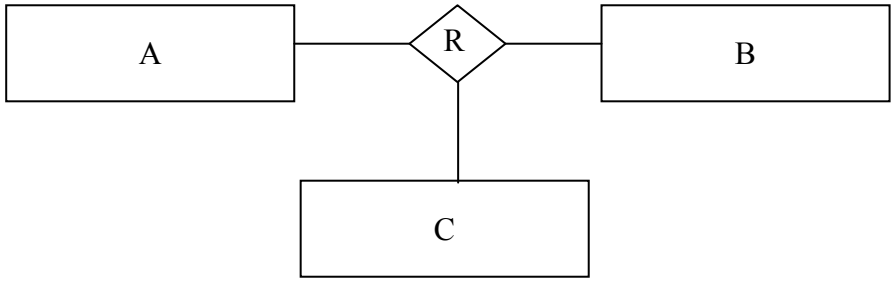
pré-requisitos $\subseteq$ { (d1, d2) / d1, d2 $\in$ Disciplina e

papel (d1) = pré-requisito e

papel (d2) = pós-requisito}

Até o momento, tratamos apenas de relacionamentos binários. Entretanto relacionamentos *n*-ários são também possíveis, ainda que muito menos corriqueiros. Um relacionamento ternário, por exemplo, só se caracteriza pelo terno, como mostra a figura 5.8. Os relacionamentos ternários normalmente são difíceis de se dar um nome e por isso é usual representá-los pelas iniciais das três entidades envolvidas, como mostra o exemplo da figura

5.9. Neste exemplo, estamos dizendo que lojas compram materiais de fornecedores, sendo que uma loja pode comprar vários materiais diferentes, de fornecedores diferentes. Já um fornecedor pode vender vários materiais para diferentes lojas. Por fim um material pode ser adquirido por várias lojas a partir de vários fornecedores.



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

Figura 5.8 – Relacionamento Ternário.

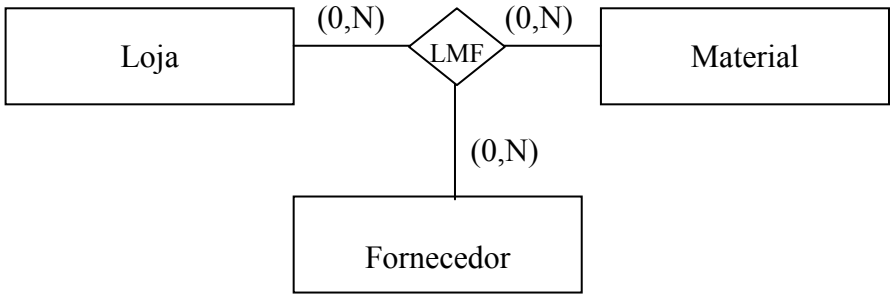
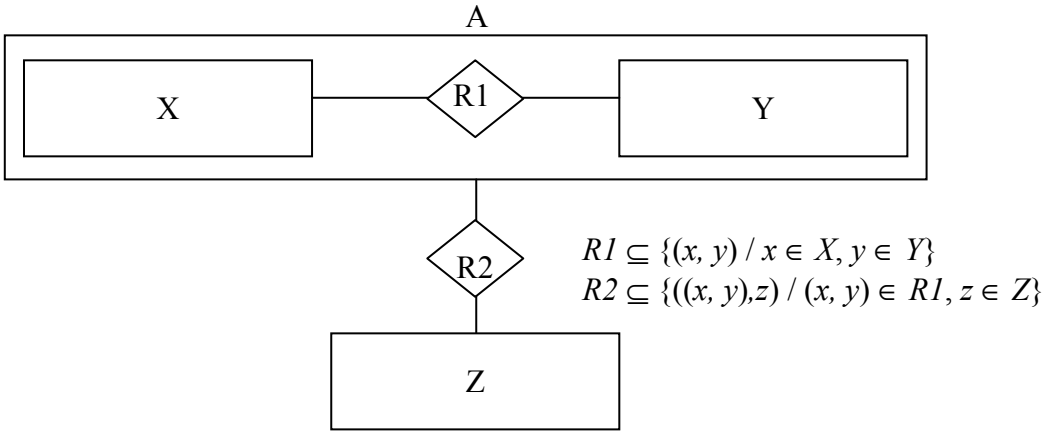


Figura 5.9 – Exemplo de Relacionamento Ternário.

Alguns relacionamentos são tão importantes que assumem o *status* de entidades. No modelo ER, esses relacionamentos são chamados de **agregação** ou **agregados**. Assim, uma agregação é uma abstração através da qual relacionamentos entre duas entidades são tratados como entidades em um nível mais alto, ou seja, um relacionamento binário R e as entidades envolvidas X e Y são considerados uma única entidade A, agregando todas as informações. Essa "nova entidade", a agregação, pode, então, relacionar-se com outras entidades do modelo, como mostra a figura 5.10.



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

Figura 5.10 – Agregação.

A figura 5.11 mostra um exemplo de agregação no contexto bancário. Nesse exemplo, o par Cliente-Conta assume o status de um Correntista, para o qual um Cartão Magnético pode ser emitido.

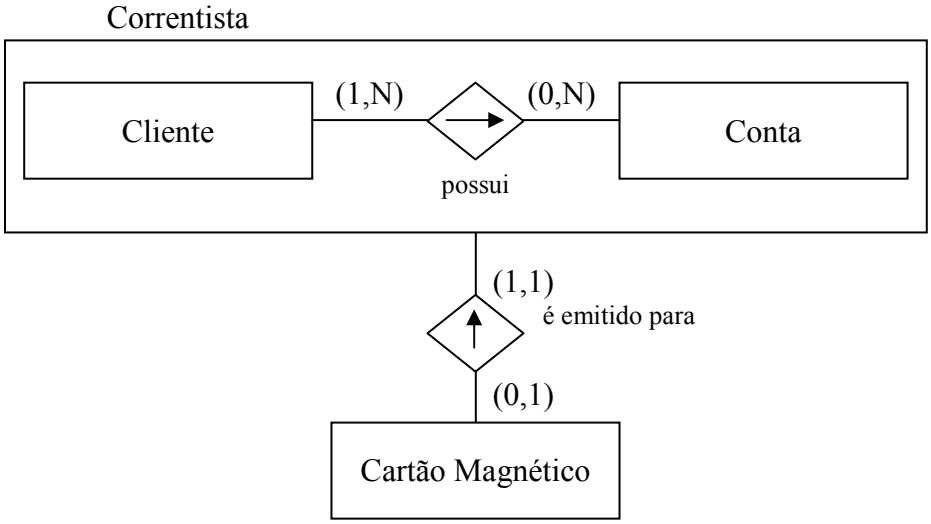


Figura 5.11 – Exemplo de Agregação.

Para prover maior facilidade na elaboração dos diagramas ER, representaremos a agregação com um retângulo envolvendo apenas o relacionamento, como mostra o exemplo da figura 5.12.

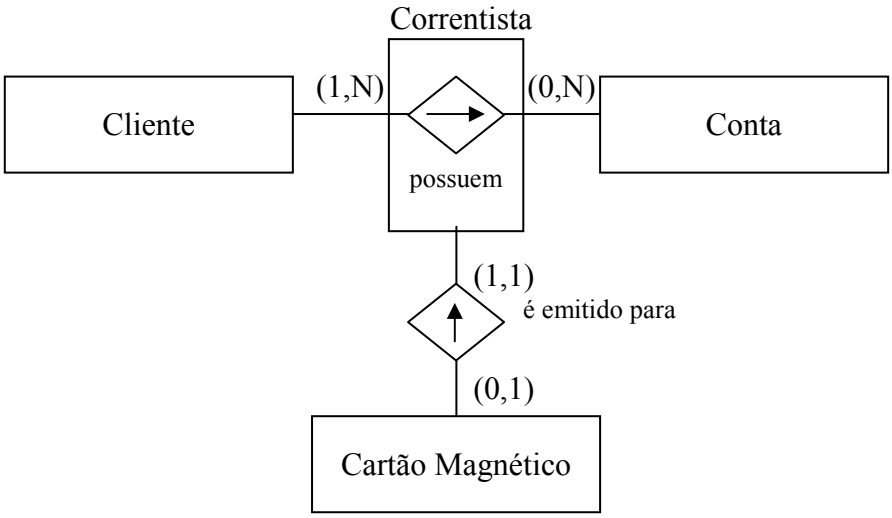


Figura 5.12 – Outra representação para Agregados.

É importante observar que agregações envolvendo relacionamentos N:1 ou 1:N, normalmente, não são necessárias. Em relacionamentos desta natureza, cada entidade cuja cardinalidade máxima é 1 (Y) só aparece no máximo uma única vez no relacionamento e, conseqüentemente, já representa o par que eventualmente possa ocorrer. Assim, as duas versões apresentadas nas figuras 5.13 e 5.14 são equivalentes quanto às informações apresentadas e preferimos utilizar sempre a versão da figura 5.14, representando agregações apenas em relacionamentos N:N.

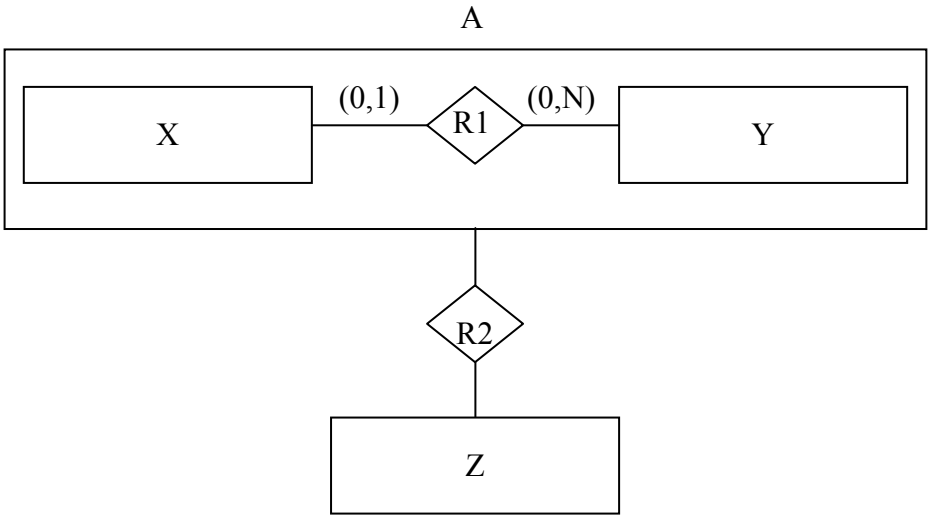


Figura 5.13 – Agregado em relacionamento 1:N.

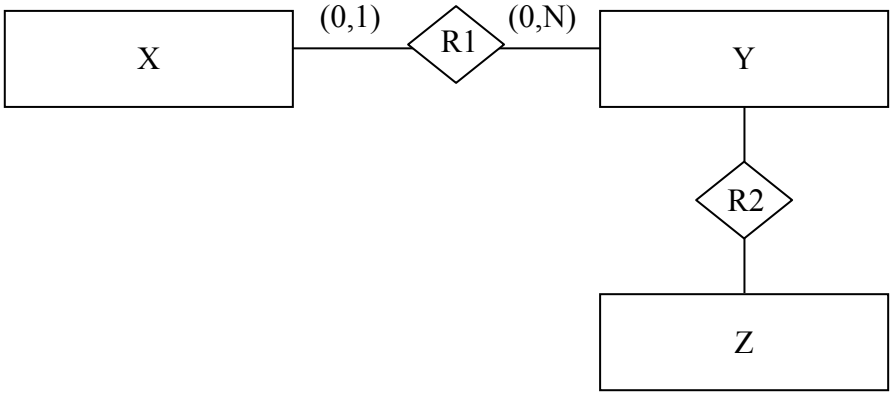


Figura 5.14 – Solução mais apropriada.

Às vezes, quer se representar um caráter opcional no que diz respeito à totalidade dos relacionamentos. Essa restrição pode ser de dois tipos:

- **Ou obrigatório**: Apenas um dos relacionamentos ocorre efetivamente, mas sempre um deles ocorre. No exemplo da figura 5.15, todo contrato é financiado (não existe contrato que não seja financiado), mas pode ser financiado ou por um banco ou por um fornecedor.

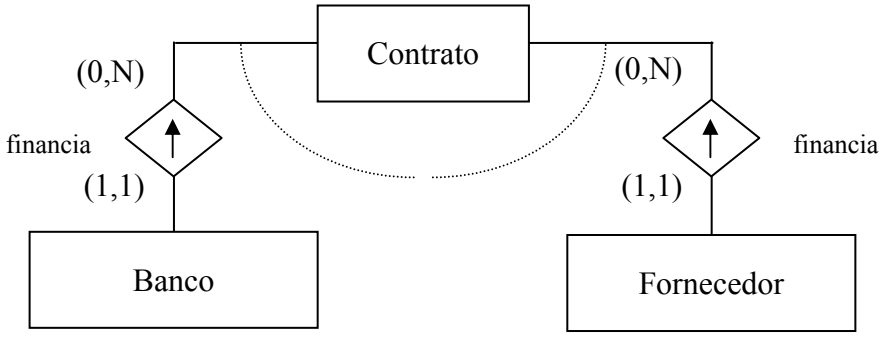


Figura 5.15 – Exemplo de Conector Ou-Obrigatório.

- **Ou opcional**: Apenas um dos relacionamentos ocorre efetivamente, mas pode ser que nenhum dos dois ocorra. No exemplo da figura 5.16, nem todo contrato é financiado. Mas se um contrato for financiado, ele será financiado ou por um banco ou por um fornecedor.

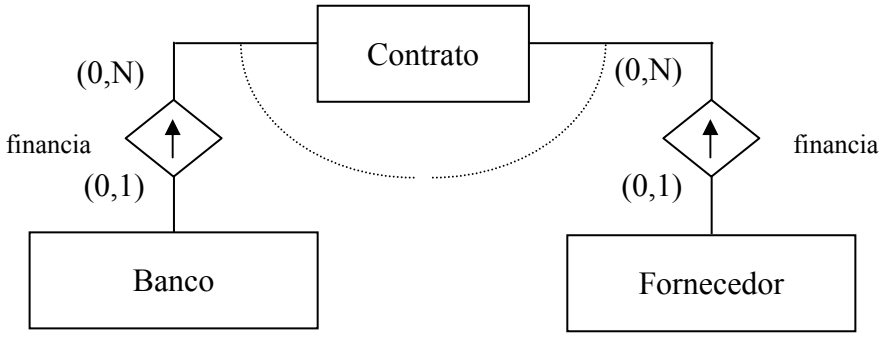


Figura 5.16 – Exemplo de Conector Ou-Opcional.

Atributos

Os atributos são utilizados para descrever características ou propriedades relevantes de entidades e relacionamentos. O conjunto de atributos de uma entidade ou relacionamento deve ser:

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

Quando um atributo pode assumir apenas um único valor, ele é dito um atributo monovalorado. Por exemplo, os atributos nome e data de nascimento de uma entidade Funcionário são monovalorados, tendo em vista que uma instância de Funcionário, por exemplo, João, possui apenas um nome e uma data de nascimento.

Por outro lado, quando um atributo pode assumir vários valores para uma mesma instância, ele é dito multivalorado. Atributos multivalorados são representados com um asterisco (*) associado. Por exemplo, o atributo telefone da entidade Funcionário é multivalorado, já que um mesmo funcionário pode ter mais de um telefone.

Atributos podem ter um valor vazio associado. Isso acontece quando para uma instância não existe um valor para aquele atributo, ou ele ainda não é conhecido. Por exemplo, o atributo telefone da entidade Funcionário pode receber um valor vazio, já que um funcionário específico pode não ter nenhum telefone, ou em um dado momento ele não ser conhecido.

Uma vez que estamos falando de conjuntos de entidades e relacionamentos, muitas vezes, é útil apontar que atributos são capazes de identificar univocamente um elemento de um conjunto. O conjunto de um ou mais atributos que identificam um elemento do conjunto é dito um atributo determinante. Atributos determinantes são sublinhados, como forma de destaque.

A figura 5.17 mostra uma representação gráfica para atributos. Ainda que essa notação possa ser empregada, de maneira geral, atributos são representados apenas no dicionário de dados para evitar modelos complexos e de difícil leitura.

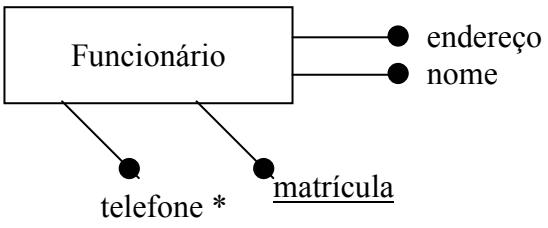


Figura 5.17 – Representação gráfica para atributos.

Vale destacar que atributos também são usados para descrever características de relacionamentos (atributos de relacionamentos) e que todas as considerações feitas até então são válidas. Atributos de relacionamentos são atributos que não são de nenhuma das duas entidades, mas sim do relacionamento entre elas e, em geral, estão relacionados com "protocolos" e datas, ou são resultantes de "avaliações", tal como no exemplo da figura 5.18.

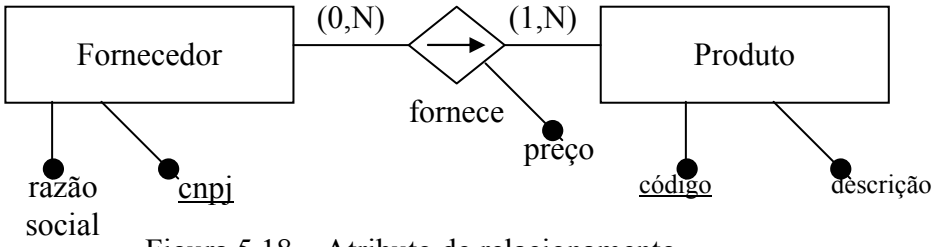


Figura 5.18 – Atributo de relacionamento.

Na prática, apenas os atributos de relacionamentos N:N são caracterizados como atributos de relacionamentos. No caso de relacionamentos 1:N ou N:1, os atributos do relacionamento podem ser perfeitamente caracterizados como atributos da entidade cuja a cardinalidade máxima é 1. No exemplo da figura 5.19, o atributo *data-de-lotação* pode ser perfeitamente caracterizado como atributo da entidade Funcionário, já que um funcionário está lotado em apenas um departamento. Logo, a data de lotação do funcionário é a data de lotação dele no departamento ao qual está associado.

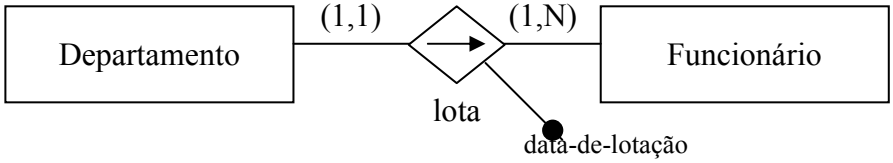


Figura 5.19 – Atributo de relacionamento caracterizado como atributo de uma das entidades.

Quando o relacionamento é N:N, há um teste que pode ser aplicado para se deduzir se um atributo é de um dos dois conjuntos de entidades ou se é do relacionamento. Seja o exemplo da figura 5.20.

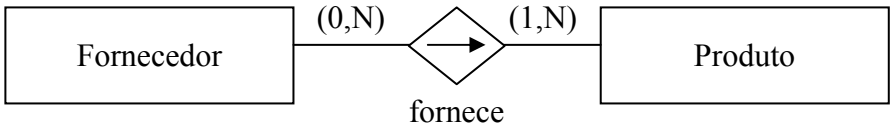


Figura 5.20 – Relacionamento N:N com atributos.

1. Fixa-se um produto, por exemplo impressora, e varia-se os fornecedores desse produto. Evidentemente o valor do atributo pode mudar. Por exemplo, a Casa do Analista vende uma impressora por R$ 350, enquanto a loja Compute vende a mesma impressora por R$ 310. Se o valor do atributo mudar ao variarmos o elemento do outro conjunto de entidades, é porque este não é atributo do primeiro conjunto de entidades, no caso Produto.
2. Procedimento análogo deve ser feito, agora, para a outra entidade. Fixando-se um fornecedor e variando-se os produtos temos: A Casa do Analista vende uma impressora por R$ 350 e um microcomputador por R$ 1.700. Como o valor do atributo variou para a mesma entidade, é sinal de que ele não é atributo de Fornecedor.
3. Se o atributo preço não é nem de Produto, nem de Fornecedor, então é um atributo do relacionamento entre os dois conjuntos de entidades, como mostra a figura 5.21.

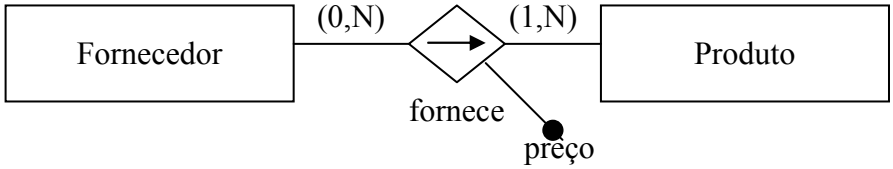


Figura 5.21 – Atributo do Relacionamento.

Uma outra questão a ser considerada relacionada a atributos é: a informação que se deseja modelar deve ser tratada como atributo de uma entidade ou como uma segunda entidade relacionada à primeira? Analisemos o seguinte exemplo: Será que departamento que oferece uma disciplina deve ser modelado como atributo da entidade Disciplina, ou merece ser uma nova entidade relacionada a ela? De forma geral, convém tratar um elemento de informação como uma segunda entidade se:

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

Podemos notar que, no exemplo, todos os critérios anteriormente enumerados foram satisfeitos e, portanto, departamento deve ser tratado como uma nova entidade, como mostra a figura 5.22.

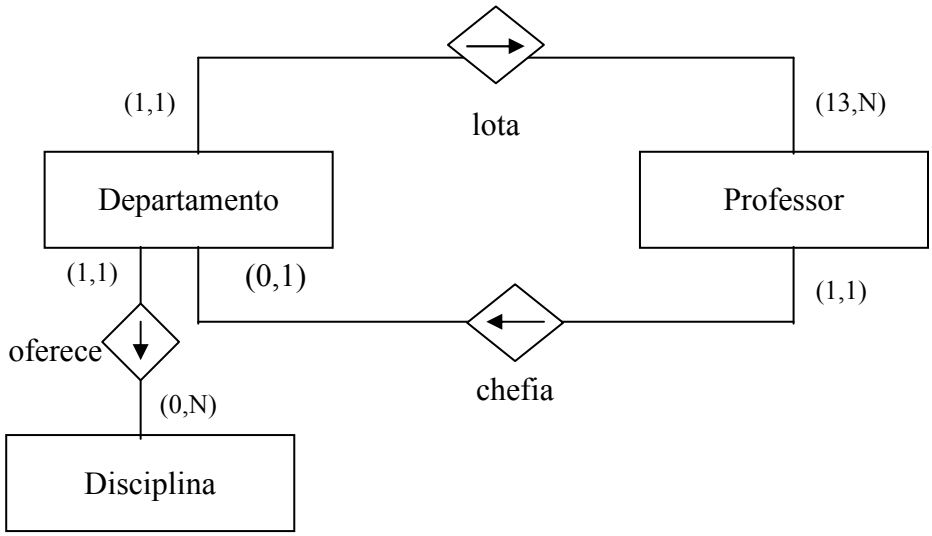


Figura 5.22 – Departamento como Nova Entidade.

Particionamento de Entidades

Muitas vezes, instâncias de entidades do mundo real se subdividem em categorias com atributos e relacionamentos parcialmente distintos. Passa a ser interessante, então, representar os atributos e relacionamentos comuns em um supertipo e os atributos e relacionamentos específicos em subtipos. Essa distinção pode ser feita por meio de:

- Generalização: uma entidade de um nível mais alto é criada, para capturar as características comuns de entidades de nível mais baixo.
- Especialização: uma entidade de nível mais alto de abstração é desmembrada em várias entidades de nível mais baixo.

A figura 5.23 mostra um exemplo de particionamento.

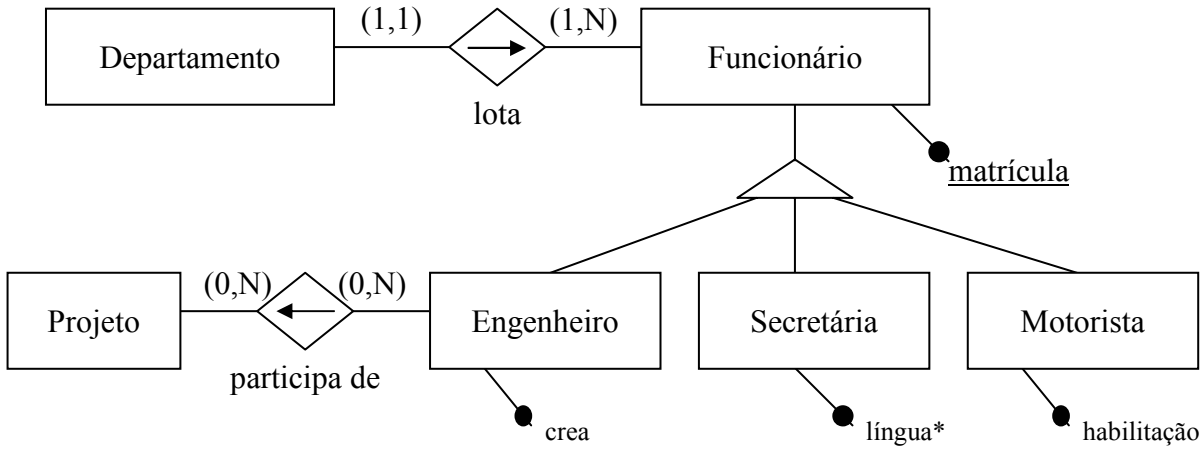


Figura 5.23 – Particionamento de um Conjunto de Entidades.

Vale a pena observar que a simbologia utilizada na representação de entidades, relacionamentos e atributos mostrada neste texto não é padronizada, isto é, não foi definido um padrão único a ser seguido por todos que utilizam o Modelo ER. Assim, diferentes textos podem utilizar simbologias diferentes.

5.3.2 - Restrições de Integridade

Muitas vezes, não somos capazes de modelar toda a estrutura de informação necessária com um diagrama ER, sobretudo no que diz respeito a restrições. Para suprir essa deficiência de representação dos diagramas ER, devemos adicionar ao modelo descrições textuais na forma de restrições de integridade, isto é, restrições do mundo real que devem ser descritas para manter a integridade do modelo. Há dois tipos básicos de restrições de integridade: restrições sobre os possíveis relacionamentos e restrições sobre os valores dos atributos.

Restrições de Integridade em Relacionamentos

Alguns relacionamentos só podem ser ocorrer se determinada restrição for satisfeita. Um exemplo de restrição de integridade são as cardinalidades. Por exemplo, se um funcionário só pode estar lotado em um único departamento, então não é possível relacionar um funcionário já lotado a um novo departamento. A cardinalidade é uma das poucas restrições de integridade que são expressas no próprio diagrama ER. Outras restrições de integridade passíveis de representação nos diagramas ER são aquelas envolvidas nos conectores ou-opcional e ou-exclusivo, conforme discutido anteriormente. Entretanto, há outras situações que não conseguimos capturar. Seja o exemplo da figura 5.24.

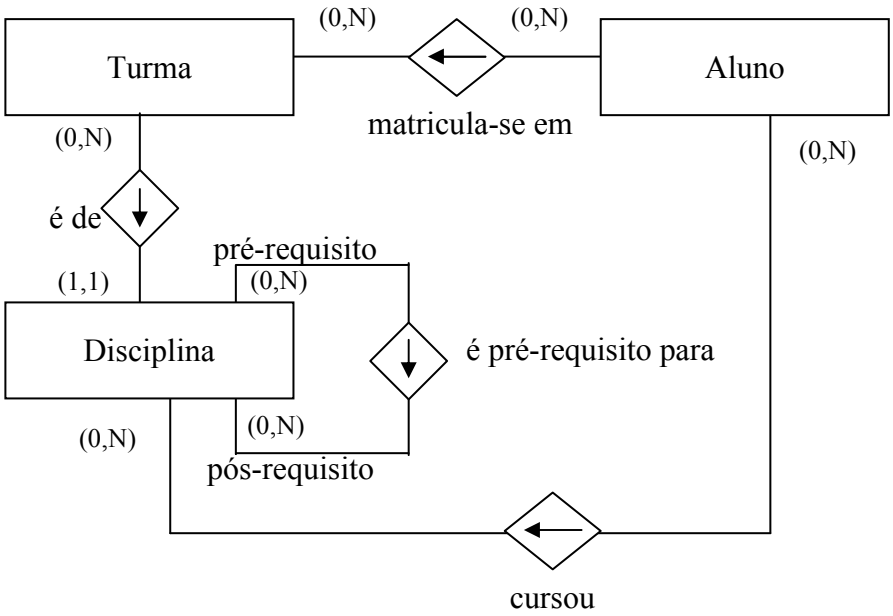


Figura 5.24 – Exemplo de auto-relacionamento.

Neste exemplo, gostaríamos de dizer, dentre outras coisas, que um aluno não pode se matricular duas vezes na mesma turma, ainda que ele possa se matricular em várias turmas. Além disso, um aluno só pode se matricular em uma turma de uma disciplina, se já tiver

cursado todos os pré-requisitos daquela disciplina. Essas duas restrições sobre possíveis relacionamentos não são passíveis de serem capturadas pela notação dos diagramas ER e devem ser, então, escritas em linguagem natural como parte do modelo ER, mais precisamente no dicionário de dados do projeto.

O primeiro tipo de restrição apontado no exemplo anterior é dito uma restrição de integridade de **repetição** e indica quantas vezes os mesmos dois elementos das entidades podem ser relacionados.

O segundo tipo é dito uma restrição de integridade de **dependência**, apontando que um relacionamento pode ser restringido por um outro relacionamento, ou depender de seus relacionamentos anteriores.

Restrições de Integridade sobre o Domínio dos Atributos

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

- **enumeração**: lista explícita de valores.

  Ex: Estado Civil : solteiro, casado, desquitado, divorciado e viúvo.

- **normas de aceitação**: regras para se identificar se o valor é válido ou não.

  Ex: Nome: qualquer conjunto de caracteres alfanuméricos, começado por uma letra.

- **intervalo**: descrição de um subconjunto de um intervalo conhecido.

  Ex: Mês: de 1 até 12.

Uma vez estabelecido o domínio, é interessante determinar valores possíveis e prováveis, isto é, alguns valores, apesar de poderem ocorrer, é pouco provável que ocorram, dependendo do contexto. Por exemplo, com relação ao atributo idade de um empregado, o valor 81 é um valor possível, mas será que ele é um valor provável, considerando que a aposentadoria ocorre de maneira compulsória aos 70 anos?

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

- **obrigatoriedade**: estabelecer se um determinado atributo pode ter um valor nulo a ele associado.

  Ex: Telefone: opcional; Nome: obrigatório.

- **dependência**: Os valores que um atributo pode assumir, muitas vezes, são dependentes dos valores de outros atributos. Neste caso é importante relacionar no dicionário de projeto como se dá esta dependência.

  Ex: O valor do atributo *dia* depende fundamentalmente do valor do atributo *mês*.

A data de demissão de um funcionário tem de ser temporalmente posterior à sua data de admissão.

5.3.3 – Dicionário de Dados

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

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

A figura 5.25 apresenta a notação adotada neste texto para elaboração de Dicionários de Dados.

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

Figura 5.25 – Notação para Dicionários de Dados.

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

(a) O cliente pode possuir um telefone.

Cliente = /*clientes da livraria*/

cpf + nome + endereço + (telefone)

(b) O cliente pode possuir mais de um telefone (ou mesmo nenhum).

Cliente = cpf + nome + endereço + {telefone}

(c) O cliente pode possuir até três telefones.

Cliente = cpf + nome + endereço + {telefone}3

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

Cliente = cpf + nome + endereço + [telefone-comercial | telefone-residencial | telefone-comercial + telefone-residencial]

5.4 - Modelagem de Estados

Diagramas de Estados são utilizados para descrever o comportamento de uma entidade ou de um relacionamento, com o objetivo de mostrar o comportamento do mesmo ao longo do seu tempo de vida. Diagramas de Estado descrevem todos os possíveis estados pelos quais uma entidade / relacionamento pode passar e as transições dos estados como resultado de eventos (estímulos) que atingem o mesmo. A figura 5.26 mostra a notação básica para diagramas de estado.

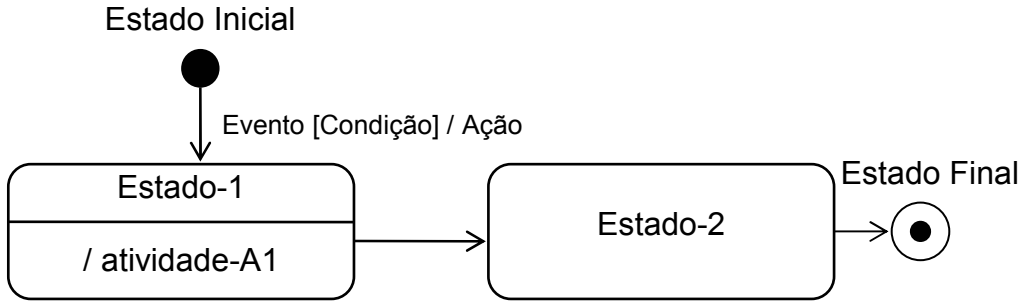


Figura 5.26 - Notação Básica para Diagramas de Estados.

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

/ atividade

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

Evento [condição] / ação

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

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

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

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

5.5 - Modelagem Funcional

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

A técnica mais difundida para esta finalidade é a utilização de Diagramas de Fluxo de Dados - DFDs, proposta por Gane e Sarson em [10] e por De Marco em [11]. Muitos outros autores citam esta técnica em suas obras, sendo que destacamos como referência [12] e [6].

Um Diagrama de Fluxo de Dados é um instrumento para a modelagem de processos, que representa um sistema como uma rede de processos, interligados entre si por fluxos de dados e depósitos de dados.

DFDs utilizam-se de quatro símbolos gráficos, visando representar os seguintes componentes: Processos, Fluxos de Dados, Depósitos de Dados e Entidades Externas. A figura 5.27 mostra a notação usada por Yourdon [6], que será a adotada neste texto. Através da utilização desses quatro componentes, podemos representar satisfatoriamente os processos e interações entre os elementos de um sistema.

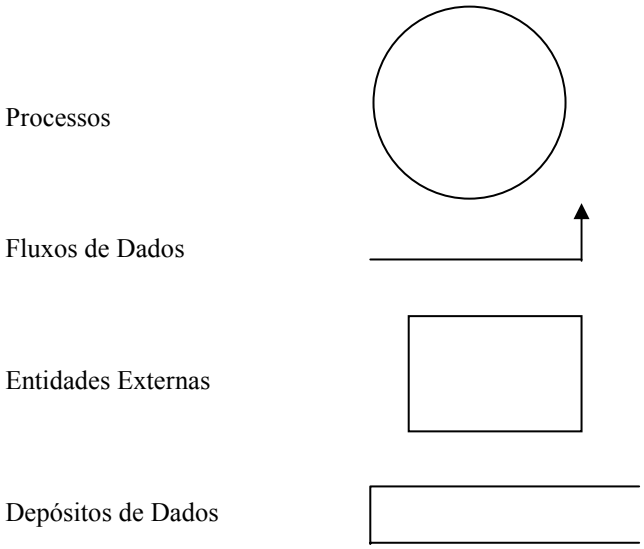


Figura 5.27 – Notação básica para construção de DFDs.

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

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

Um DFD mostra as fronteiras do sistema: aquilo que não for uma Entidade Externa será interno ao sistema, delimitando assim a fronteira do sistema. Além disso, mostra todas as relações entre dados (armazenados e que fluem no sistema) e os processos que manipulam e transformam esses dados, encarnando totalmente a filosofia do paradigma estruturado.

Processos

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

- transformações do conteúdo de um dado de entrada no conteúdo de um dado de saída, sem armazenamento interno no sistema (figura 5.28);

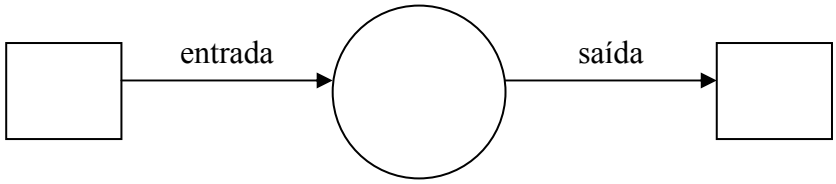


Figura 5.28 – Transformações de dados.

- inserções ou modificações do conteúdo de dados armazenados, a partir do conteúdo (possivelmente transformado) de dados de entrada, como mostra a figura 5.29;

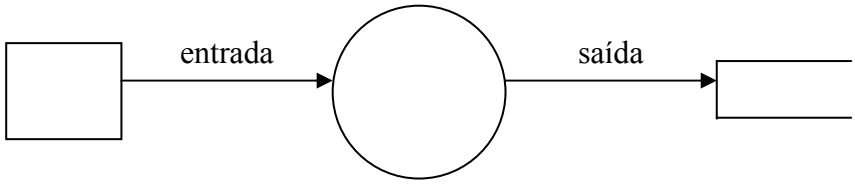


Figura 5.29 – Armazenamento de dados.

- transformações de dados previamente armazenados no conteúdo de um dado de saída, como mostra a figura 5.30.

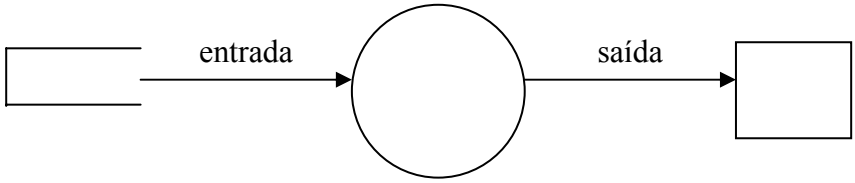


Figura 5.30 – Geração de dados de saída a partir de dados armazenados.

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

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

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

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

As especificações das lógicas dos processos só devem ser feitas para processos simples. Processos complexos devem ser decompostos em outros processos, até se atingir um nível de reduzida complexidade. Esta descrição não deve ser confundida com o detalhamento

integral da lógica do processo que deverá ser feito na fase de projeto, mas deve servir de base para essa outra etapa.

Uma heurística para se definir se um processo merece ou não ser representado em um DFD é dada em função da descrição de sua lógica. Quando essa descrição utilizar aproximadamente uma ou duas páginas, então o processo parece estar adequado. Processos descritos em três ou quatro linhas são simples demais para serem tratados como processos em um DFD. Por outro lado, se a descrição da lógica do processo necessitar de quatro ou mais páginas, então esse processo está muito abrangente e não deve ser tratado como um único processo, mas sim deve ser decomposto em processos de menor complexidade. Para situações desta natureza, duas técnicas são utilizadas: fissão ou explosão, como estudaremos mais à frente.

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

Fluxos de Dados

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

Um fluxo de dado em um DFD pode ser considerado como um caminho através do qual poderão passar uma ou mais estruturas de dados em tempo não especificado. Note que em um DFD não se representam elementos de natureza não informacional, isto é, dinheiro, pessoas, materiais etc.

Devemos observar se um fluxo de dados entra e sai de um processo sem modificação. Isso representa uma falha, haja visto que um processo transforma dados. Embora possa parecer um tanto óbvio, é bom lembrar que um mesmo conteúdo pode ter diferentes significados em pontos distintos do sistema e, portanto, os fluxos devem ter nomes diferentes. No DFD da figura 5.31, um mesmo conjunto de informações sobre um cliente tem significados diferentes quando passa pelos fluxos *dados-novo-cliente* e *dados-cliente*. No primeiro caso, os dados ainda não foram validados e, portanto, podem ser válidos ou inválidos, enquanto, no segundo fluxo, esses mesmos dados já foram validados.

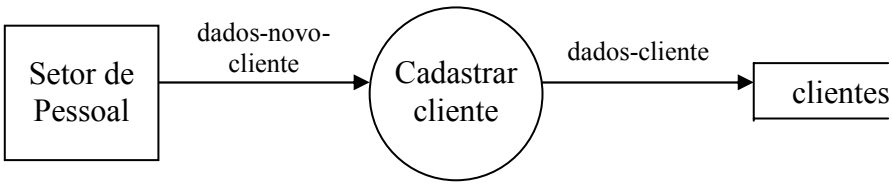


Figura 5.31 – Mesmo conteúdo de dados em fluxos diferentes.

Fluxos de dados que transportam exatamente o mesmo conteúdo de/para um depósito de dados, não precisam ser nomeados. No exemplo da figura 5.31, se o fluxo *dados-cliente* apresentar exatamente o mesmo conteúdo do depósito *clientes*, não há necessidade de nomeá-lo, como mostra a figura 5.32.

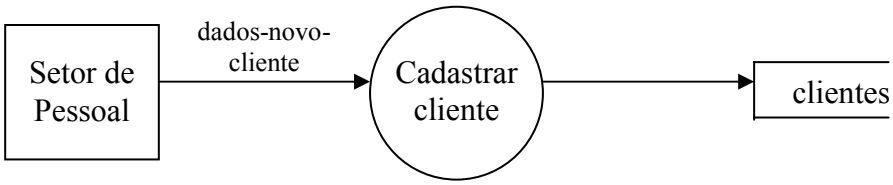


Figura 5.32 – Fluxo de dados não nomeado.

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

Setas ramificadas significam que o mesmo fluxo de dados está indo de uma fonte para dois destinos diferentes, isto é, cópias do mesmo pacote de dados estão sendo enviadas para diferentes partes do sistema, como mostra a figura 5.33.

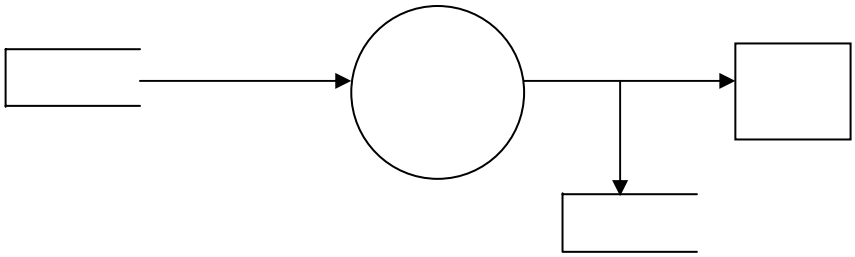


Figura 5.33 – Fluxo ramificado.

Quando for necessário cruzar fluxos de dados em um DFD, devemos utilizar um arco, como mostra a figura 5.34.

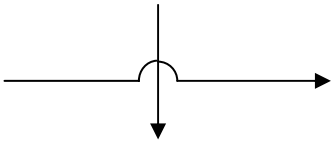


Figura 5.34 – Fluxos de dados que se cruzam em um diagrama.

É importante realçar que DFDs não indicam a seqüência na qual fluxos de dados entram ou saem de um processo. Essa seqüência é descrita apenas na especificação do processo.

Depósitos de Dados

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

Um depósito de dados é representado por um retângulo sem a linha lateral direita, com um nome e um identificador (opcional) em seu interior. Às vezes, para evitar o cruzamento de linhas de fluxos de dados ou para impedir que longas linhas de fluxos de dados saiam de um lado para outro do diagrama, um mesmo depósito de dados pode ser representado mais de uma vez no diagrama. Nessa situação, adicionamos uma linha vertical na lateral esquerda do retângulo, como mostra a figura 5.35.

Figura 5.35 – Notação para depósitos de dados.

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

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

A semântica dos acessos aos depósitos de dados é mostrada na figura 5.36.

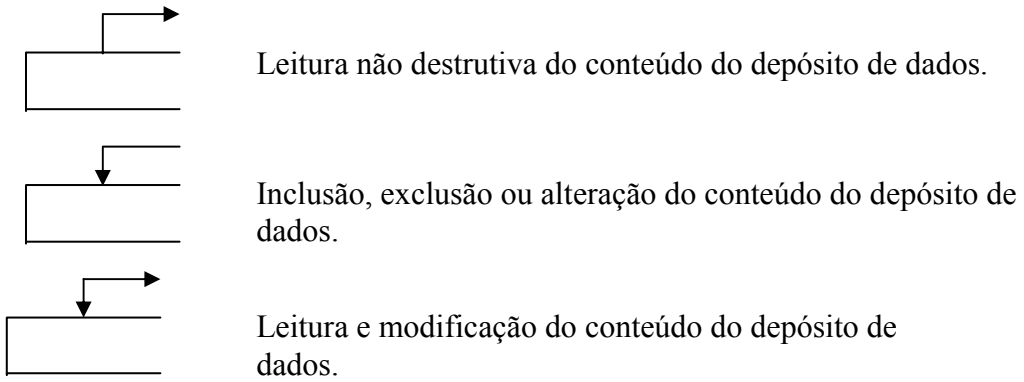


Figura 5.36 – Semântica dos acessos a depósitos de dados em um DFD.

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

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

Muitas vezes, diferentes sistemas compartilham uma mesma base de dados e, portanto, vários sistemas poderão estar lendo e atualizando os conteúdos de um mesmo depósito de

dados. É interessante mostrar este fato explicitamente no DFD e, nesse caso, podemos notar três situações distintas:

- O sistema em questão apenas lê as informações do depósito de dados, não sendo responsável por qualquer alteração de seu conteúdo. Neste caso, utilizamos a notação da figura 5.37.

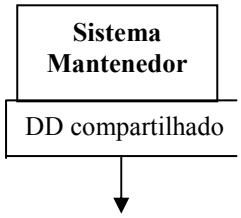


Figura 5.37 – Sistema apenas acessando depósito de dados mantido por outro sistema.

- O sistema em questão apenas gera as informações que são utilizadas por outros sistemas. Representamos essa situação segundo a notação da figura 5.38.

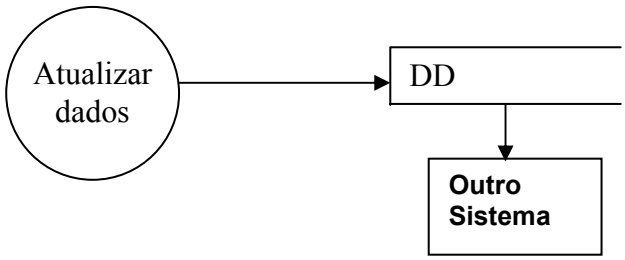


Figura 5.38 – Sistema atualizando dados utilizados por outro sistema.

- Ambos os sistemas atualizam o depósito de dados. A notação para esta situação é mostrada na figura 5.39.

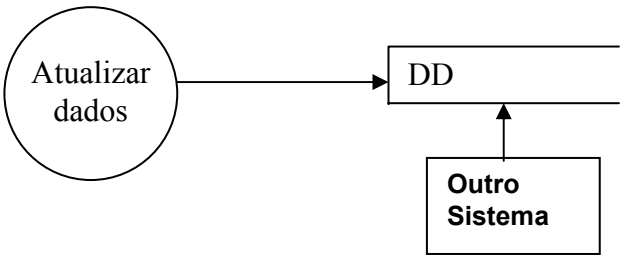


Figura 5.39 – Ambos os sistema atualizando dados de um mesmo depósito.

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

- Observar se todos os elementos de dados que fazem parte do depósito têm como efetivamente chegar lá, isto é, fazem parte de pelo menos um fluxo de dados que chega ao depósito.

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

Entidades Externas

Entidades externas ou terminadores são fontes ou destinos de dados do sistema. Representam os elementos do ambiente com os quais o sistema se comunica. Tipicamente, uma entidade externa é uma pessoa (p.ex. um cliente), um grupo de pessoas (p. ex. um departamento da empresa ou outras instituições) ou um outro sistema que interaja com o sistema em questão. Uma entidade externa deve ser identificada por um nome e representada por um retângulo. Assim como no caso dos depósitos de dados, em diagramas complexos, podemos desenhar um mesmo terminador mais de uma vez, para se evitar o cruzamento de linhas de fluxos de dados ou para impedir que longas linhas de fluxos de dados saiam de um lado a outro do diagrama. Nesse caso, convenciona-se utilizar um traço diagonal no canto inferior direito do símbolo da entidade externa, como mostra a figura 5.40.

Figura 5.40 – Notações para representar entidades externas.

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

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

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

5.4.1 - Relações entre DFDs e DERs

Conforme discutido anteriormente, depósitos de dados são representações em um DFD para entidades e relacionamentos em um modelo ER. Entretanto, em um DFD, não há uma representação explícita dos relacionamentos entre entidades. Para indicar que o relacionamento entre entidades existe, a representação dos acessos dos processos aos depósitos de dados deve obedecer à seguinte regra geral: ao criar ou excluir um relacionamento ou uma entidade que participa de um relacionamento, mostre o acesso aos depósitos de dados que correspondem ao relacionamento e às entidades que participam do relacionamento. A figura 5.41 mostra a representação gráfica desses acessos.

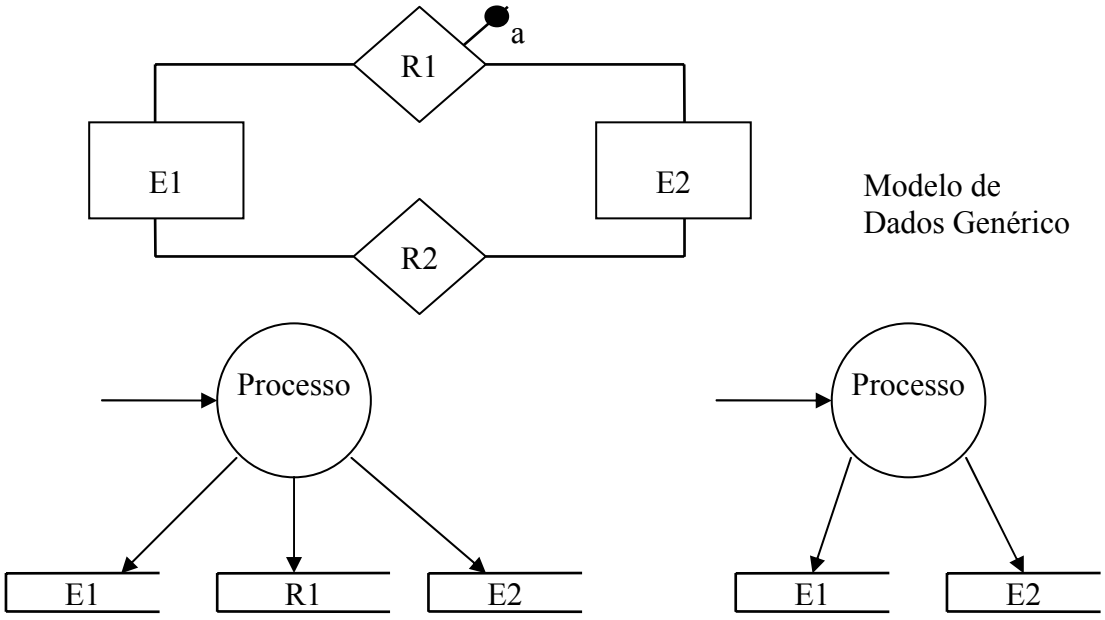


Figura 5.41 – Acessos a depósitos de dados.

No caso do relacionamento R1, como esse relacionamento tem um atributo (a), ele é representado em um DFD como sendo um depósito de dados. Assim, para criar ou excluir uma ocorrência de R1, representam-se acessos a R1, E1 e E2. Já o relacionamento R2, como esse não possui atributos, não dá origem a um depósito de dados. Para criar ou excluir uma ocorrência de R2, são representados acessos a E1 e E2.

5.4.2 - Construindo DFDs

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

Fissão

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

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

Explosão

O processo original permanece no diagrama, sendo criado um novo DFD de nível inferior, consistindo de processos menos complexos. Assim, um projeto não é representado

por um único DFD, mas sim por um conjunto de DFDs em vários níveis de decomposição funcional.

Quando a explosão é utilizada, alguns aspectos importantes devem ser observados. O primeiro deles diz respeito ao número de níveis que devem ser esperados em um sistema. A priori, esse número não deve ser pré-fixado, mas lembre-se que o número total de processos cresce exponencialmente quando se passa de um nível para o imediatamente inferior.

Tipicamente são quatro os níveis de representação:

- **C – Contexto**: mostra o sistema como uma "caixa-preta", trocando informações (fluxos de dados) com entidades externas ao sistema. Define o escopo de abrangência do sistema, indicando que se está renunciando à possibilidade de examinar qualquer coisa além da fronteira definida pelas entidades externas. É parte integrante do modelo ambiental, segundo a Análise Essencial.
- **0 (Zero) – Geral ou de Sistema**: é uma decomposição do diagrama de contexto, mostrando o funcionamento do sistema em questão, isto é, as grandes funções do sistema e as interfaces entre elas. Os processos nesse diagrama recebem os números 1, 2, 3 etc. É necessário assegurar a coerência entre os diagramas **C** e **0**, isto é, assegurar que os fluxos de dados entrando ou saindo do diagrama **0** efetivamente reproduzem as entradas e saídas do diagrama **C**. No diagrama **0**, devem aparecer os depósitos de dados necessários para a sincronização dos processos.
- **N – Detalhe**: Uma diagrama de detalhe representa a decomposição de um dos processos do diagrama **0** e, portanto, é nomeado com o número e o nome do processo que está sendo detalhado. A princípio, deverão ser elaborados diagramas **N** para os processos do diagrama **0** que sejam complexos e, portanto, careçam de decomposição. É necessário resguardar a coerência entre o diagrama **0** e cada diagrama detalhado elaborado. Os processos em um diagrama **N** são numerados com o número do processo que está sendo detalhado (p. ex., 2) e um número seqüencial, separados por um ponto (p. ex., 2.1, 2.2, etc.).
- **E – Detalhe Expandido**: um diagrama deste tipo representa a decomposição de um dos processos do diagrama **N**. Esse nível de decomposição pode vir a ser necessário caso um processo do nível **N** ainda for muito complexo. Esse nível pode ser desdobrado sucessivamente até se atingir o grau necessário de simplicidade. Entretanto, se muitos níveis forem necessários, cuidado! Provavelmente, o contexto funcional da aplicação (diagrama de contexto) está muito abrangente e merece revisão.

Fissão ou Explosão?

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

Recomendações para a Construção de DFDs

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

5.4.3 - Técnicas de Especificação de Processos

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

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

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

```
se A < B
    então TOTAL ← B - A;
    senão TOTAL ← A + B;
fim-se;
```

- Uso de comparativos como: Maior que / Menor que, Mais de / Menos de.
  Seja a seguinte declaração: "Até 20 unidades, sem desconto. Mais de R$20, 5% de desconto".
  E exatamente 20 unidades, que tratamento deve ser dado?

- Ambigüidades do E/OU.
  Seja a seguinte declaração: "Clientes que gerarem mais de um milhão de reais em negócios por ano **e** possuírem um bom histórico de pagamentos **ou** que estiverem conosco há mais de 20 anos, devem receber tratamento prioritário".
  Quem deverá receber tratamento prioritário? Clientes com mais de 1 milhão em negócios por ano que possuírem bom histórico de pagamentos? Clientes com mais de 20 anos? Clientes com mais de 1 milhão e (ou bom histórico, ou mais de 20 anos)?
  Note que pela declaração não fica claro quando deverá ser aplicado o tratamento prioritário.

- Uso de Adjetivos Indefinidos
  Na declaração do item anterior, o que é um ***bom*** histórico de pagamentos? Devemos tomar cuidado ao utilizarmos adjetivos indefinidos. Quando o fizermos, devemos tomar o cuidado de defini-los.

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

- Português Estruturado
- Tabelas de Decisão
- Árvores de Decisão
- Combinação das técnicas acima

Português Estruturado

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

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

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

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

```
se      <condição>
          então grupo_ de_ações_1;
          senão grupo_de_ações_2;
 fim-se;
```

Exemplo:

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

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

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

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

1. **para cada "X" faça**
   grupo_de_ações;
   fim-para;

   Exemplo:

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

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

   grupo_de_ações;

   fim-enquanto;

   Exemplo:

   ```
   enquanto existir notas faça
           ler nota;
           consistir dados;
   fim-enquanto;
   ```

3. **repita**

   grupo_de_ações;

   até que <condição seja verdadeira>;

   Exemplo:

   ```
   repita
          ler nota;
          consistir dados;
   até que todas as notas tenham sido processadas;
   ```

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

- deve ser clara, concisa, completa e livre de ambigüidades;
- todos os dados citados na especificação que estejam definidos no dicionário de dados devem estar em itálico, incluindo depósitos de dados;
- os dados definidos localmente são escritos em fonte normal;
- sempre que um comando de seleção ou repetição for utilizado, os comandos do bloco interno (grupo_de_ações) devem estar identados, de modo a dar a clareza de que esses comandos fazem parte das ações da seleção ou repetição.

Árvores de Decisão

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

Como podemos notar pelo exemplo da figura 5.42, uma Árvore de Decisão é muito boa para representar a lógica decisória. Entretanto, se for necessário descrever a lógica de um processo como um conjunto de instruções, combinando decisões e ações intermediárias, a árvore de decisão deve ser preterida em favor do português estruturado ou combinada a ele.

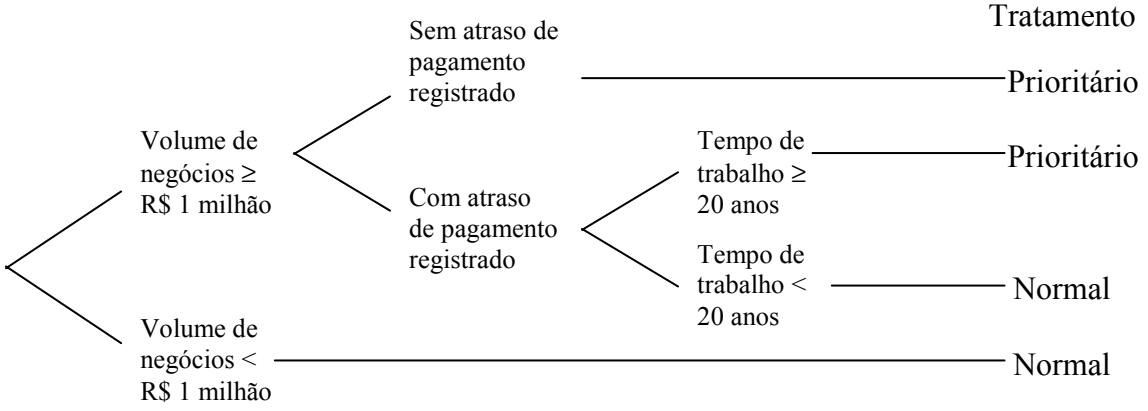


Figura 5.42 – Exemplo de Árvore de Decisão.

Tabelas de Decisão

Tabelas de decisão são usadas em aplicações semelhantes às das árvores de decisão. As árvores de decisão são mais indicadas, quando o número de decisões for pequeno e nem todas as combinações de condições forem possíveis. As tabelas de decisão aplicam-se melhor a situações em que o número de ações é grande e ocorrem muitas combinações de condições.

Também devemos utilizar tabelas de decisão se existirem dúvidas de que a árvore de decisão não mostra toda a complexidade do problema.

O formato básico de uma tabela de decisão é mostrado na figura 5.43.

Nome da Tabela	
Condições	Combinações
Ações	Regras

Figura 5.43 – Formato básico de uma Tabela de Decisão.

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

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

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

- Tabela de Entrada Limitada: os valores de uma condição se limitam a dois. Exemplos típicos deste tipo de tabelas são as tabelas cujas condições são escritas sob a forma de perguntas, de modo que as respostas sejam “sim” ou “não”, como mostra o exemplo da figura 5.44.

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

Figura 5.44 – Tabela de Entrada Limitada.

- Tabela de Entrada Ampliada: Uma condição pode ter mais de dois valores possíveis diferentes, como no exemplo da figura 5.45.

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

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

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

Peso: Leve (L): ≤ 100kg; Pesado (P): > 100Kg

Figura 5.45 – Tabela de Entrada Ampliada.

Muitas vezes, grupos de condições levam à mesma ação. Para estes casos, podemos utilizar tabelas compactadas, como a do exemplo 5.46.

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

Figura 5.46 – Tabela Compactada.

Quando uma única tabela se tornar muito grande ou complexa, podemos utilizar tabelas encadeadas, onde uma tabela faz referência a outras, como mostra o exemplo da figura 5.47.

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

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

Figura 5.47 – Tabelas Encadeadas.

5.4.4 – Modelagem Funcional com DFDs e a Análise Essencial

Quando empregamos a filosofia da Análise Essencial na modelagem funcional, um DFD contendo um único processo é construído para cada um dos eventos listados na lista de eventos. Caso o evento seja complexo demais e mereça ser decomposto em outros processos, então as técnicas de fissão ou explosão devem ser aplicadas. Construídos os DFDs para os eventos específicos, os mesmos podem ser agrupados, dando origem a DFDs de nível superior, até se chegar a um DFD de nível 0 e, por fim, a um DFD de Contexto.

Contudo, é importante ressaltar que a maior parte dos eventos em uma lista de eventos pode ser simples e que representar tais eventos por meio de DFDs pode não trazer ganhos substanciais para o desenvolvimento. Muito pelo contrário: pode gerar uma quantidade desnecessária de DFDs, aumentando muito a documentação do sistema, com pouca utilidade. Assim, recomendamos a elaboração de DFDs apenas para os eventos da lista que sejam mais complexos e que estejam intimamente ligados ao propósito do sistema. Para os demais, as descrições dos eventos providas no modelo ambiental são suficientes.

Referências Bibliográficas

1. R.S. Pressman, *Engenharia de Software*, Rio de Janeiro: McGraw Hill, 5ª edição, 2002.
2. I. Sommerville, *Engenharia de Software*, São Paulo: Addison-Wesley, 6ª edição, 2003.
3. S.L. Pfleeger, *Engenharia de Software: Teoria e Prática*, São Paulo: Prentice Hall, 2ª edição, 2004.
4. D.F.Togneri, "Apoio Automatizado à Engenharia de Requisitos Cooperativa", Dissertação de Mestrado, Mestrado em Informática da UFES, 2002.

5. S. Pompilho. *Análise Essencial: Guia Prático de Análise de Sistemas*. IBPI Press, Editora Infobook, Rio de Janeiro, 1995.
6. E. Yourdon. *Análise Estruturada Moderna*. Editora Campus, 1990.
7. C.M.S. Xavier, C. Portilho. Projetando com Qualidade a Tecnologia de Sistemas de Informação. Livros Técnicos e Científicos Editora, 1995.
8. P. Chen. Gerenciando Banco de Dados: A Abordagem Entidade-Relacionamento para Projeto Lógico. McGraw-Hill, 1990.
9. W. Setzer. *Bancos de Dados*. 2ª Edição, Editora Edgard Blücher, 1987.
10. C. Gane, T. Sarson. *Análise Estruturada de Sistemas*. Livros Técnicos e Científicos Editora, 1983.
11. T. De Marco. Análise Estruturada e Especificação de Sistemas. Editora Campus, 1983.
12. C. Gane. *Desenvolvimento Rápido de Sistemas*. Livros Técnicos e Científicos Editora, 1988.

6 – Projeto

Referências: [1] Cap. 13, [2] Cap. 10, [3] Cap. 5.

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 atividade 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 atividade de projeto envolve a modelagem de como o sistema será implementado, com a adição dos requisitos não funcionais aos modelos construídos na análise, como ilustra a figura 6.1. Assim, o objetivo do projeto é incorporar a tecnologia aos requisitos essenciais do usuário, projetando o que será construído na implementação. Para tal, é necessário conhecer a tecnologia disponível e as facilidades do ambiente de software no qual o sistema será implementado.

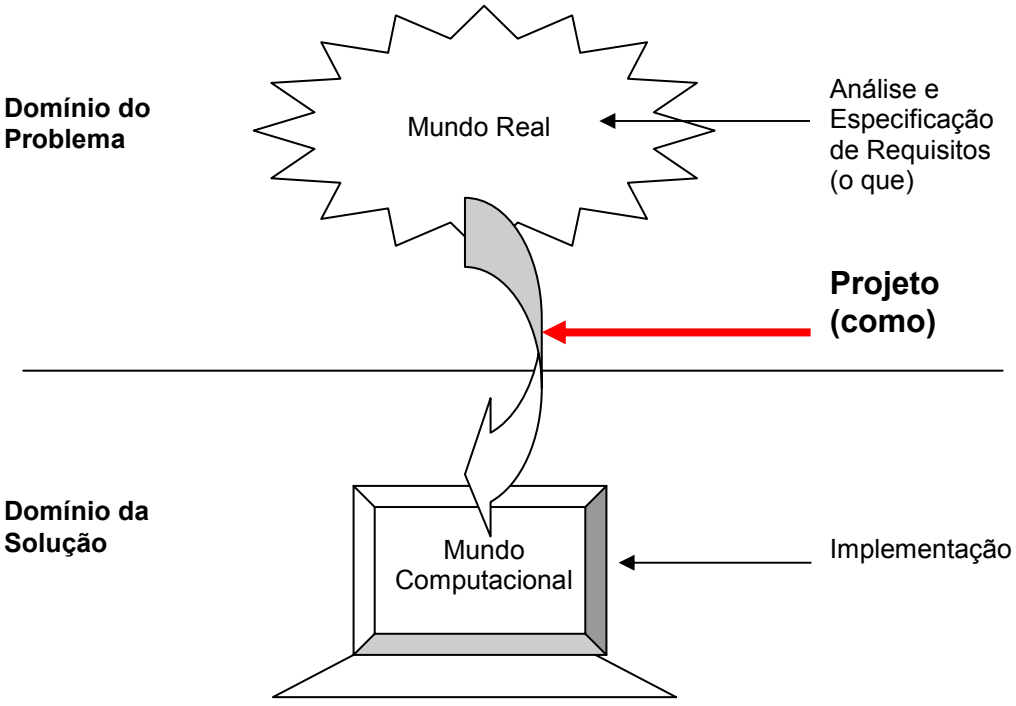


Figura 6.1 – Visão Geral da Atividade de Projeto.

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.

No projeto de sistemas, um modelo de projeto é tipicamente gerado a partir dos modelos de análise, com o objetivo de representar o que deverá ser codificado na fase de implementação. Independentemente do paradigma adotado, o projeto deve produzir:

- Projeto da Arquitetura do Software: visa a definir os grandes componentes estruturais do software e seus relacionamentos.
- Projeto de Dados: tem por objetivo projetar a estrutura dos dados necessária para implementar o software.
- Projeto de Interfaces: descreve 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: tem por objetivo refinar e detalhar a descrição procedimental dos componentes estruturais da arquitetura do software.

A seguir, cada uma dessas sub-atividades do projeto de sistemas é discutida à luz do paradigma estruturado.

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 essa plataforma de implementação.

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

Em um modelo de dados relacional, os conjuntos de dados são representados por tabelas de valores. Cada tabela 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:

- Entidades e agregados devem dar origem a tabelas;
- Uma instância de uma entidade ou de um agregado deve ser representada como uma linha da tabela correspondente;
- Um atributo de uma entidade 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 esse 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 6.2, 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 6.3.
- 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 6.4.
- Caso contrário, é melhor transpor a chave que dará origem a uma coluna mais densa, isto é, que terá menos valores nulos.

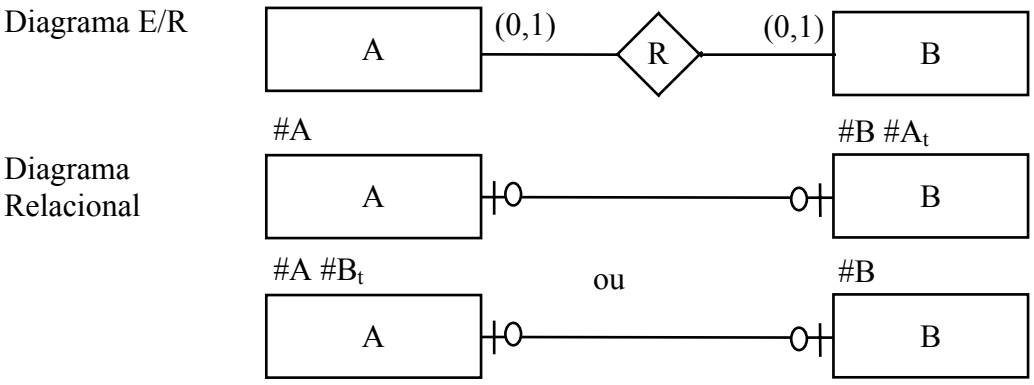


Figura 6.2 - Tradução de Relacionamentos 1:1 do Diagrama E/R para o Relacional.

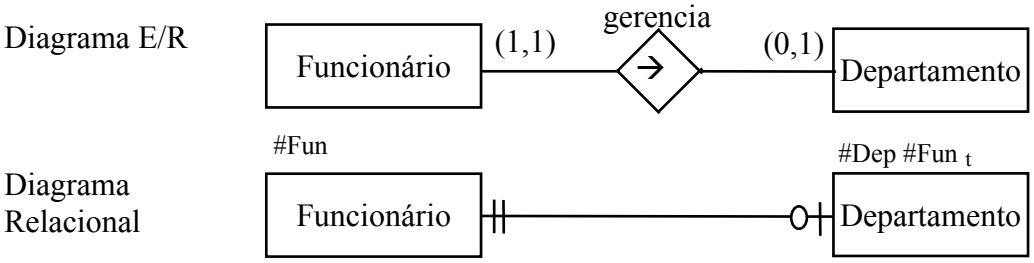
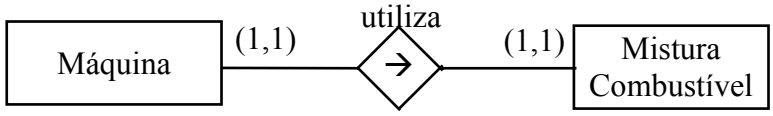


Figura 6.3 – Exemplo de Relacionamento 1:1.

Diagrama E/R



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

Diagrama Relacional

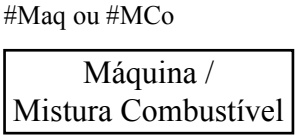


Figura 6.4 - Exemplo de um relacionamento 1:1 total em ambos os lados.

Relacionamentos 1 : N

Neste caso, deve-se transpor a chave da tabela correspondente à entidades de cardinalidade máxima N para a tabela que representa a entidade cuja cardinalidade máxima é 1, como mostra a figura 6.5.

Diagrama E/R

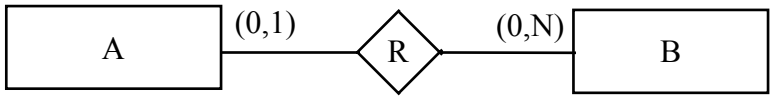


Diagrama Relacional

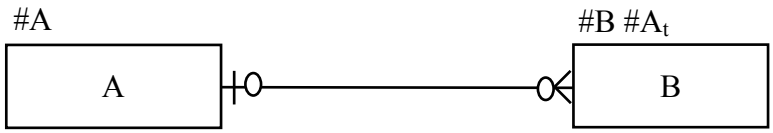


Figura 6.5 - Tradução de Relacionamentos 1:N do Diagrama E/R para o Relacional.

Um *A* pode estar associado a vários *B*s, mas um *B* só pode estar associado a um *A*, logo se deve transpor a chave primária de *A* para *B*. A figura 6.6 mostra um exemplo desta situação.

Diagrama E/R

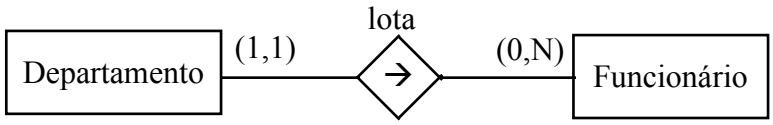


Diagrama Relacional

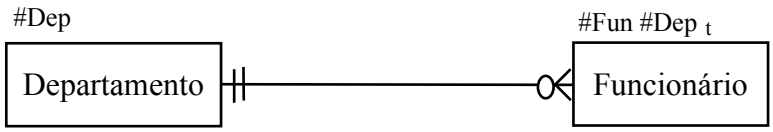


Figura 6.6 – Exemplo de um relacionamento 1:N.

Relacionamentos N : N

No caso de relacionamentos N:N (agregados ou não), deve-se criar uma terceira tabela, transpondo as chaves primárias das duas tabelas que participam do relacionamento N:N, como mostra a figura 6.7. Se existirem atributos do relacionamento (agregado), esses 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 6.8.

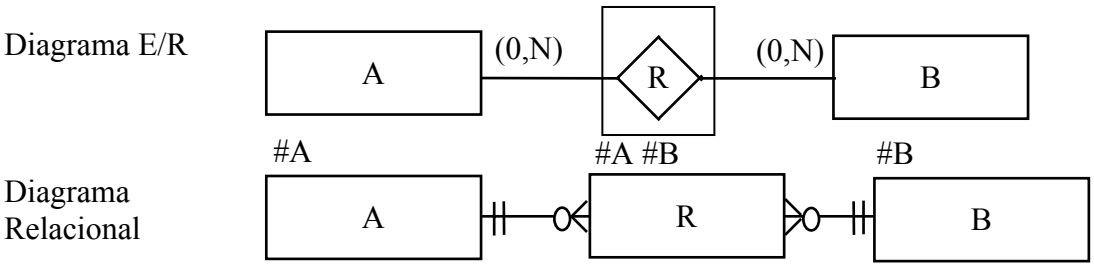


Figura 6.7 - Tradução de Relacionamentos N:N do Diagrama E/R para o Relacional.

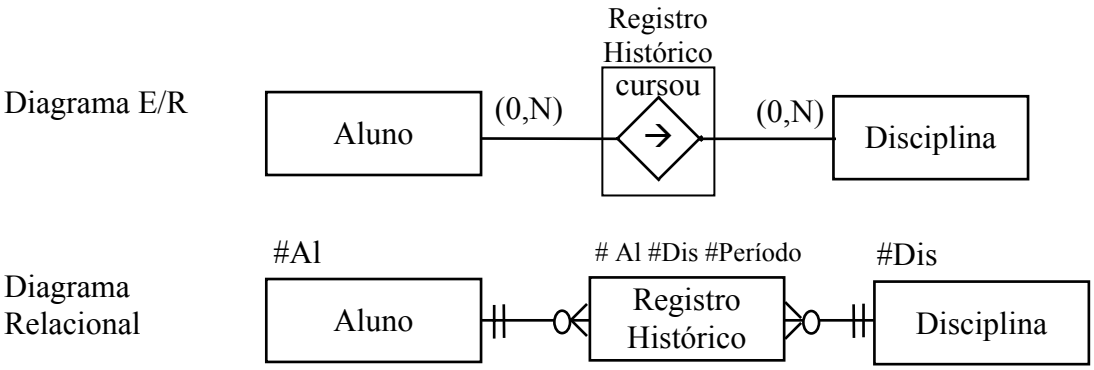


Figura 6.8 – Exemplo de relacionamento N:N.

Auto-Relacionamentos

Os auto-relacionamentos devem seguir as mesmas regras de tradução de relacionamentos, como mostram os exemplos das figuras 6.9 e 6.10.

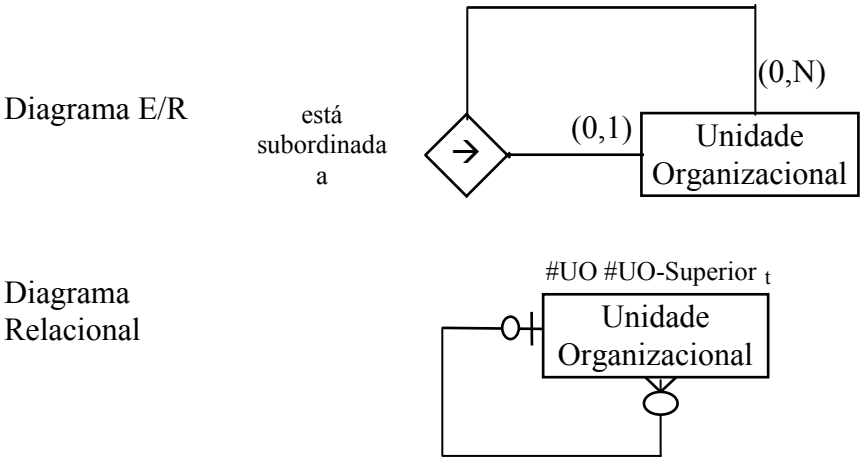


Figura 6.9 – Exemplo de auto-relacionamento 1:N.

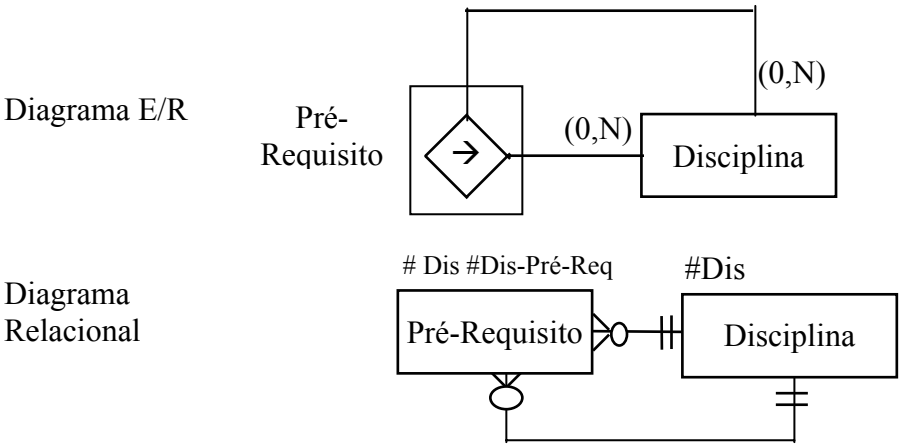


Figura 6.10 – Exemplo de auto-relacionamentos N:N.

Relacionamentos entre uma Entidade 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 6.11.

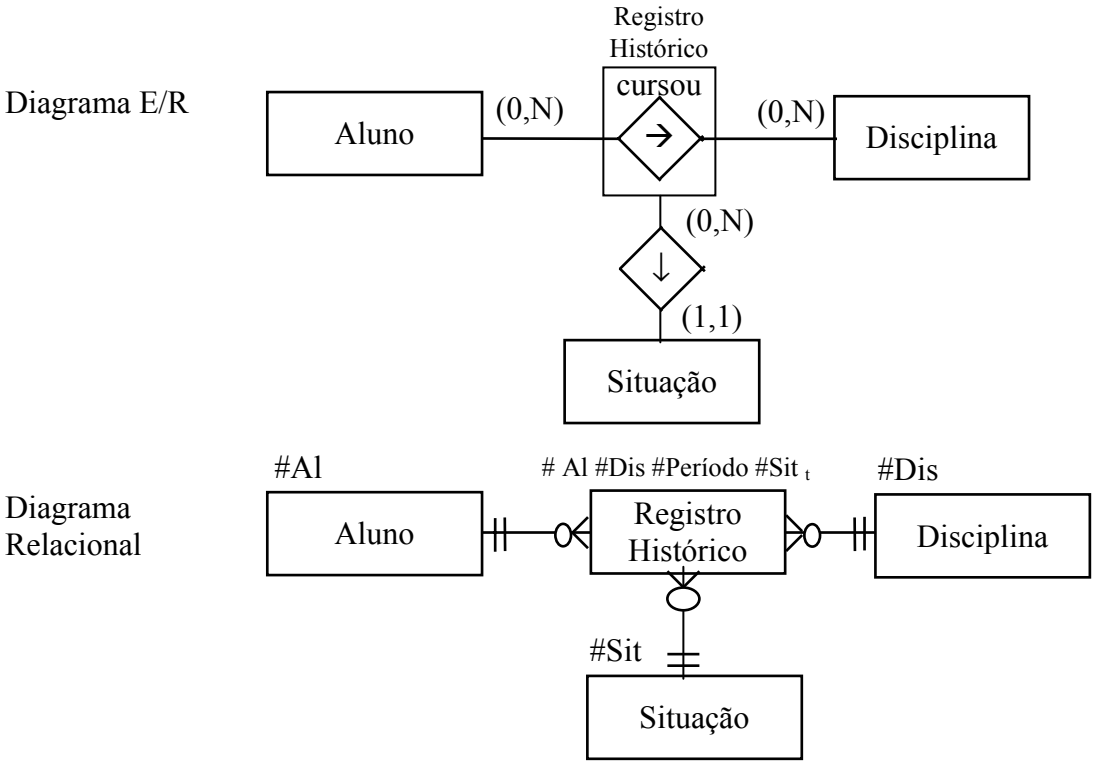


Figura 6.11 – Exemplo de relacionamento 1:N entre uma entidade e um agregado.

Relacionamento Ternário

No caso de relacionamentos ternários, deve-se criar uma nova tabela contendo as chaves das três entidades envolvidas, como mostra a figura 6.12. Assim como no caso de agregados, se existirem atributos do relacionamento, esses deverão ser colocados na nova tabela. Caso seja necessário, algum desses atributos pode ser designado para compor a chave primária da nova tabela.

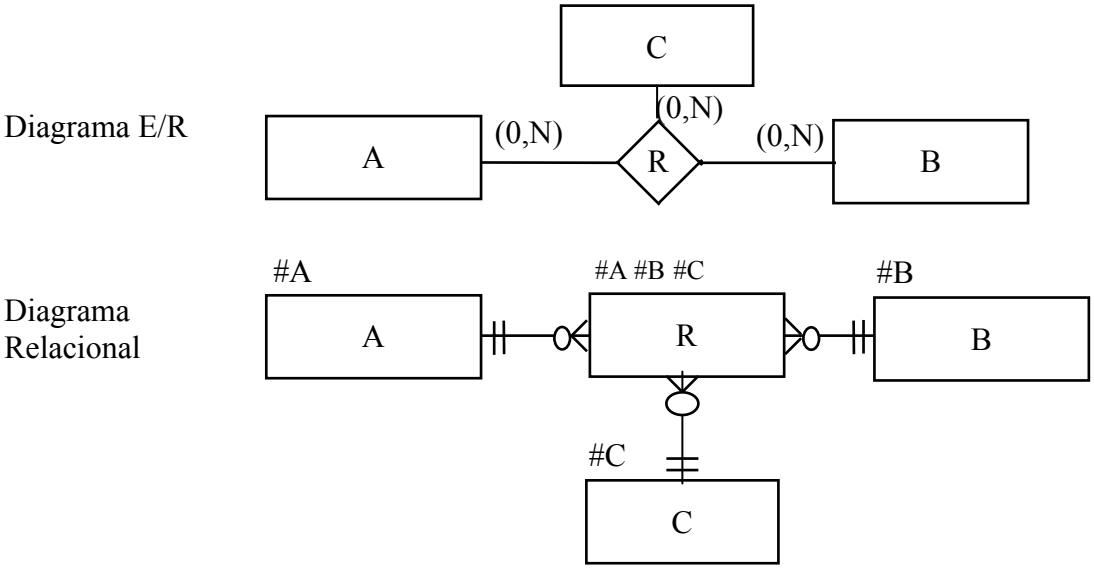


Figura 6.12 - Tradução de Relacionamentos Ternários.

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 6.13. 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.

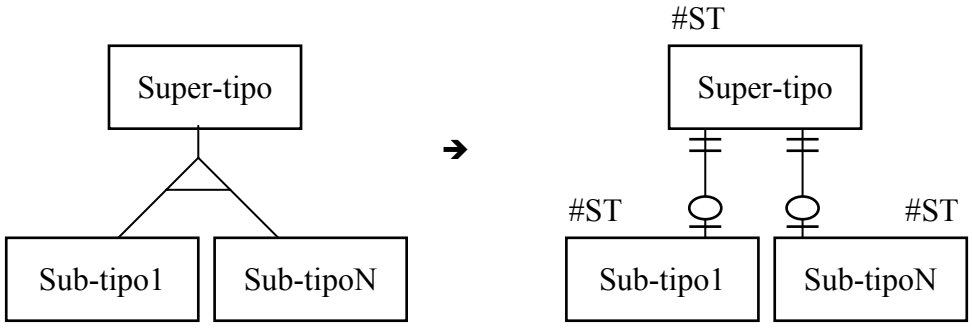


Figura 6.13 – Tradução de Particionamento.

Atributos Multivalorados

Segundo a propriedade do modelo relacional que nos diz que cada célula de uma tabela 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. Essa 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 6.14.

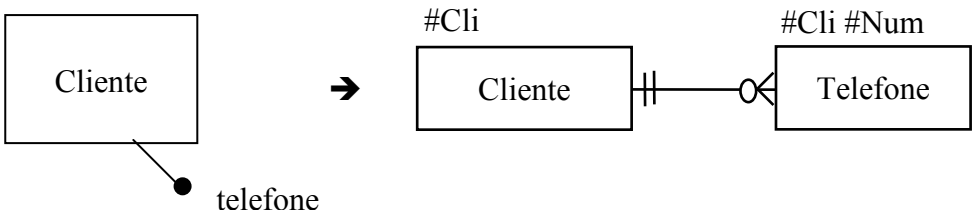


Figura 6.14 – Mapeamento Geral de Atributos Multi-valorados.

6.2 - 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). Nessa 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 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? Essas são apenas algumas das muitas questões que devem ser levantadas durante o projeto da interface com o usuário [1]. De maneira geral, o projeto de interfaces com o usuário segue o seguinte processo global, como mostra a figura 6.15:

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 dos 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 (não está habituado a usar computadores ou mecanismos específicos de interação com os sistemas computacionais) 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 necessárias para utilizar bem a interface;

- *Instruído e freqüente*: possui bom conhecimento da aplicação e domina bem os mecanismos de interação. Geralmente, usuários desse tipo 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 prototipagem 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).

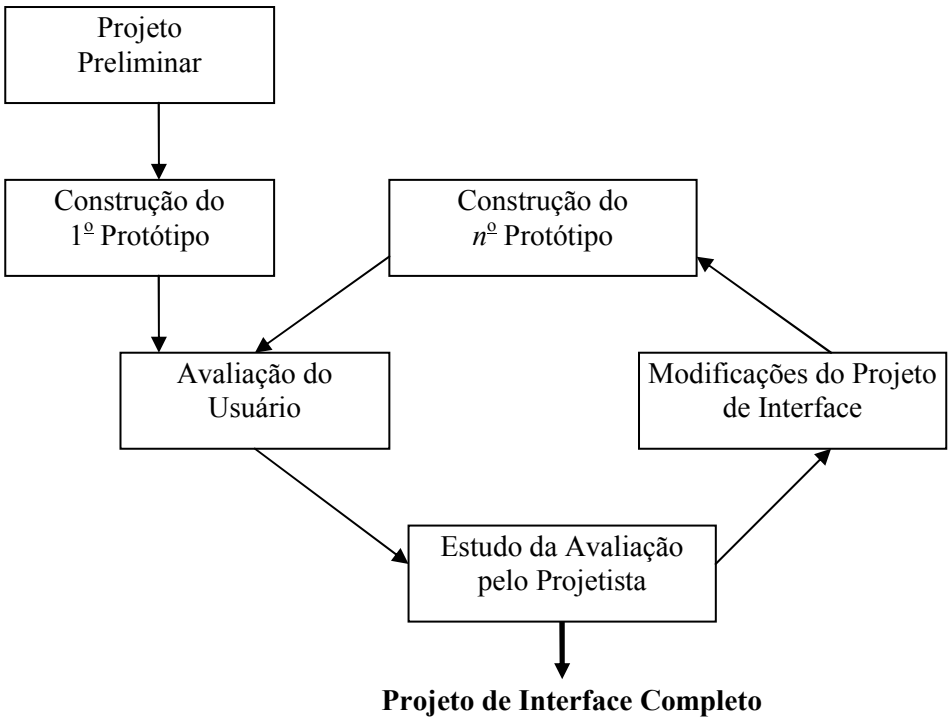


Figura 6.15 - Abordagem Iterativa para o Projeto de Interface com o Usuário.

6.3 - 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, estratégicas e heurísticas capazes de conduzir a bons projetos de programas. O objetivo é desenvolver programas com menor complexidade, usando o princípio "dividir para conquistar". 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.

O projeto estruturado de sistemas, em sua dimensão de funções, considera que o projeto de programas envolve duas grandes etapas: o projeto da arquitetura do sistema e o projeto detalhado dos módulos. Em ambos os casos, técnicas de Projeto Modular de Programas são empregadas. Apesar de usar diferentes variações para o projeto arquitetural e para o projeto detalhado, 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**: Indica a forma como os módulos interagem entre si.

O bloco básico de construção de um programa estruturado é, portanto, um módulo. Assim, os modelos do projeto estruturado de programas são organizados como uma hierarquia de módulos. A idéia básica é estruturar os programas em termos de módulos e conexões entre esses módulos.

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

- Procura solucionar sistemas complexos através da divisão do sistema em "caixas pretas" (os módulos) e pela organização dessas "caixas pretas" em uma hierarquia conveniente para uma implementação computadorizada.
- Utiliza ferramentas gráficas, o que tornam mais fácil a 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.

O Projeto Modular procura simplificar um sistema complexo, dividindo-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 tampouco seus procedimentos internos, para podermos utilizá-la.

Sistemas compostos por caixas pretas são facilmente construídos, testados, corrigidos, entendidos e modificados. Desse modo, o primeiro passo no controle da complexidade no projeto estruturado consiste em dividir um sistema em módulos, de modo a atingir as seguintes metas:

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

Organizando Módulos Hierarquicamente

Antes de iniciarmos uma discussão sobre Projeto Modular de Programas, passemos a observar os exemplos das figuras 6.16, 6.17 e 6.18, que mostram três organogramas de empresas.

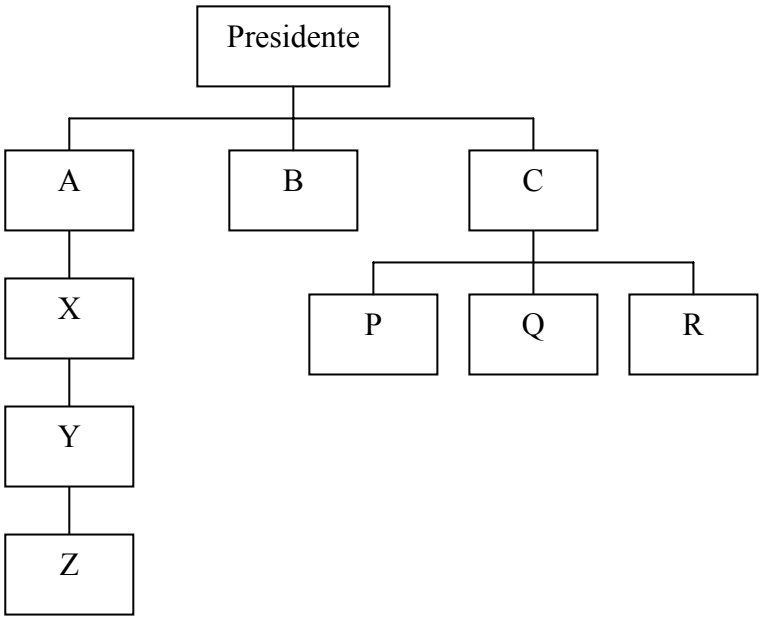


Figura 6.16 - Organograma da Empresa 1.

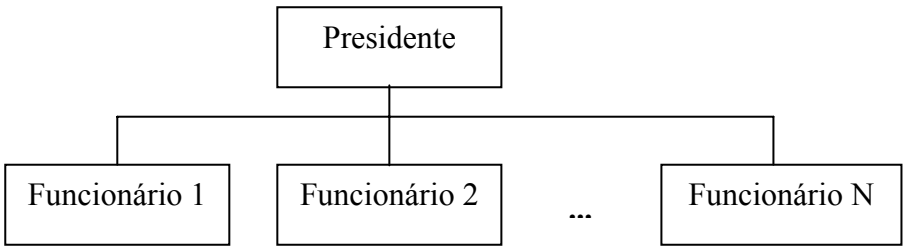


Figura 6.17 - Organograma da Empresa 2.

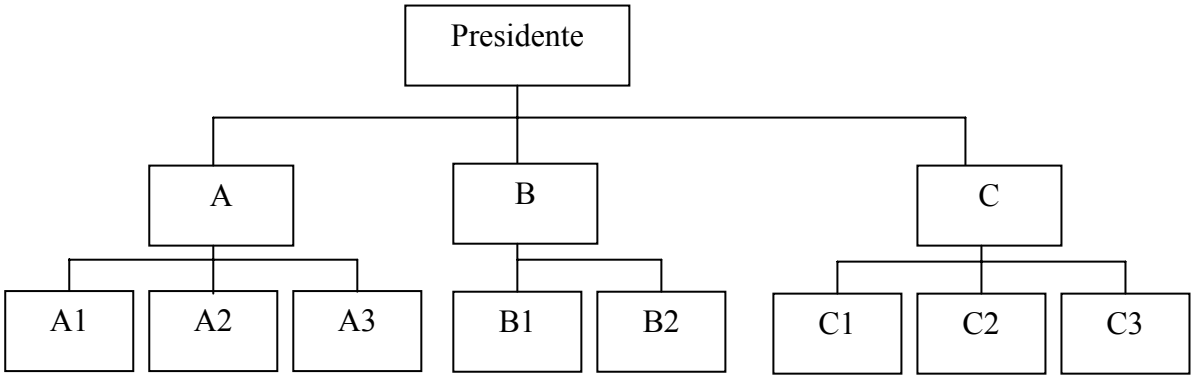


Figura 6.18 - Organograma da Empresa 3.

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 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 [4]. 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. Essas 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.

6.3.1 - 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 [4]. 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 deve procurar espelhar a estrutura do negócio que o sistema está tratando. A descrição do escopo, com sua subdivisão em sub-sistemas e módulos, e a lista de eventos e descrições associadas devem ser a base para a construção dos DHFs.

Cada executável deve dar origem a um DHF. As funcionalidades controladas por esse executável devem ser tratadas como módulos-filhos do módulo inicial do diagrama. Funções menores que compõem uma macro-função podem ser representadas como módulos-filhos do módulo correspondente. Para sistemas de médio a grande porte, contudo, representar todas as funcionalidades em um único diagrama pode torná-lo muito complexo. Assim, novos DHFs podem ser elaborados para agrupar certas funcionalidades.

Tomemos como exemplo um sistema de entrega em domicílio de refeições, cujo escopo é o seguinte:

- Subsistema Controle de Cardápio, envolvendo macro-funções para: Cadastrar Refeições, Sobremesas e Bebidas. Cada uma dessas macro-funções teria funções para incluir, excluir, alterar e consultar esses diferentes tipos de itens de cardápio;
- Subsistema Atendimento a Clientes, envolvendo macro-funções para Cadastrar Cliente, Controlar Pedido e Consultar Cardápio. Assim como os demais cadastros, o cadastro de clientes teria funções para incluir um novo cliente, alterar dados de cliente, consultar e excluir clientes. Já o controle de pedidos envolveria funções para efetuar um novo pedido, alterar dados de pedido, cancelar pedido, definir entregador e registrar atendimento de pedido. Por fim a consulta ao cardápio teria funções para consultar refeições, sobremesas e bebidas.

Com base nesse escopo e considerando que cada subsistema deve ser implementado como uma aplicação executável, poderíamos construir o DHF mostrado na figura 6.19. Nesse diagrama, optou-se por não representar os módulos-filhos do módulo *Controlar Pedido*, uma vez que ele é bastante complexo, com vários sub-módulos, o que traria uma complexidade indesejada para o DHF. Assim, além do diagrama da figura 6.19, um outro, cujo módulo inicial seria *Controlar Pedido*, deveria ser elaborado.

Vale ressaltar que 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.

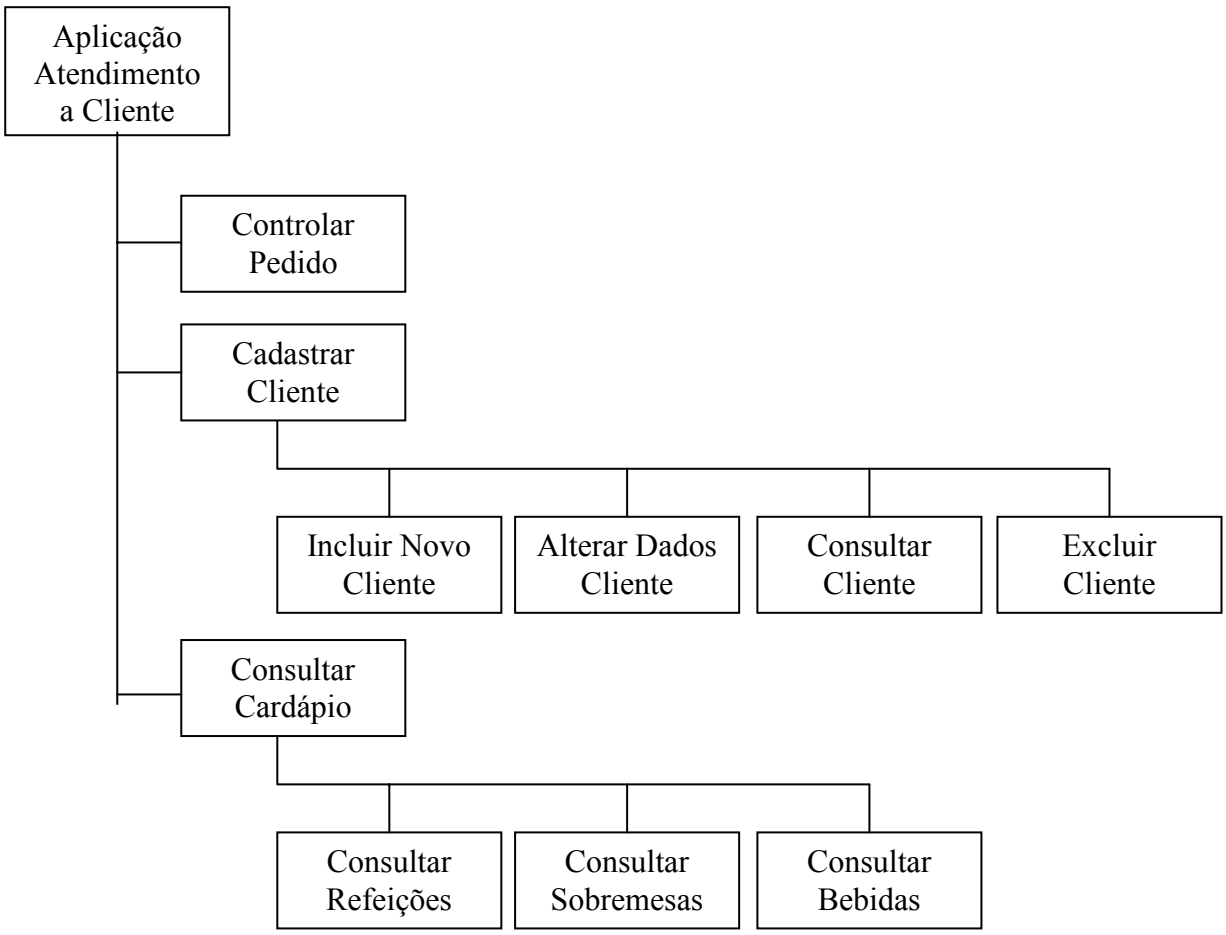


Figura 6.19 – DHF para o subsistema de Atendimento a Clientes do Sistema de Entrega de Refeições.

6.3.2 - 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 [4].

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 esses 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, tabelas de decisão e árvores de decisão, discutidos no capítulo 5.

Assim sendo, um DEM mostra:

- A divisão 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.

Notação Utilizada na Elaboração de DEMs

A seguir, são apresentadas as principais notações utilizadas para elaborar Diagramas de Estrutura Modular [5]:

- **Módulo**: Em um DEM, um módulo é representado por um retângulo, dentro do qual está contido seu nome, como mostra a figura 6.20. Um módulo pré-definido é aquele que já existe em uma biblioteca de módulos e, portanto, não precisa ser descrito ou detalhado.

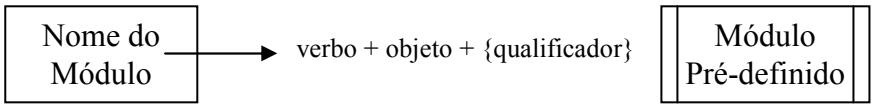


Figura 6.20 – Simbologia para Módulos em um DEM.

- **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 6.21, 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.

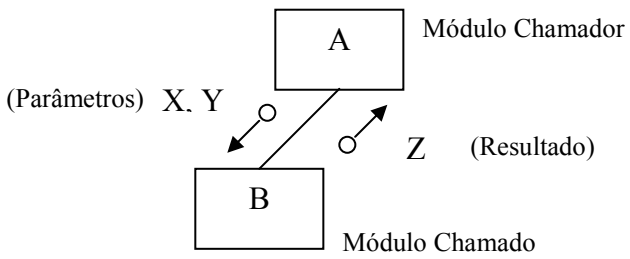


Figura 6.21 – Conexão entre módulos.

- **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 6.22 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.

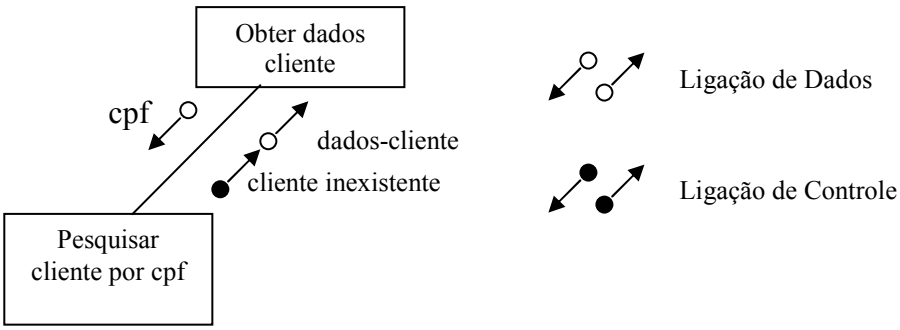


Figura 6.22 – Comunicação entre módulos.

- **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 6.23. 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*.

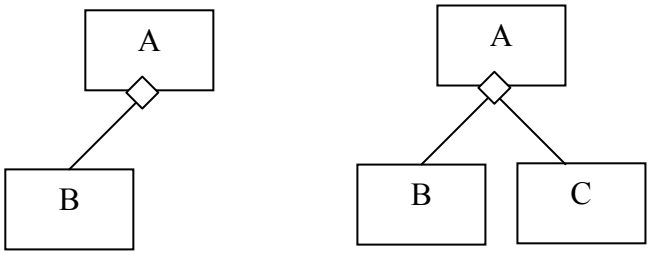


Figura 6.23 – Chamada Condicional.

- **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 6.24. No exemplo, os módulos *B* e *C* são chamados repetidas vezes pelo módulo *A*.

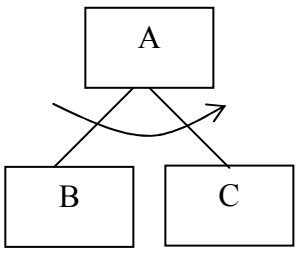


Figura 6.24 – Chamada Iterativa.

- **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 6.25.

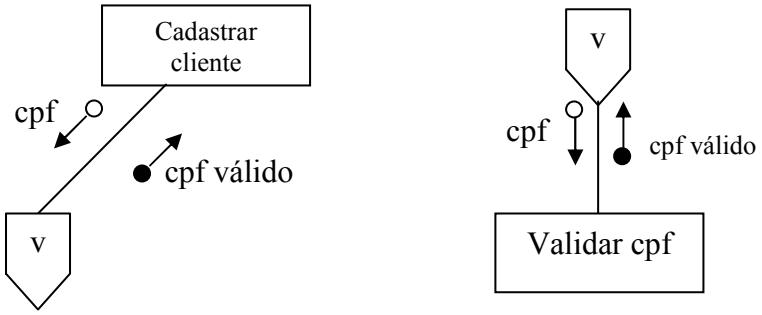


Figura 6.25 – Conectores.

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 [4].

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 6.26.

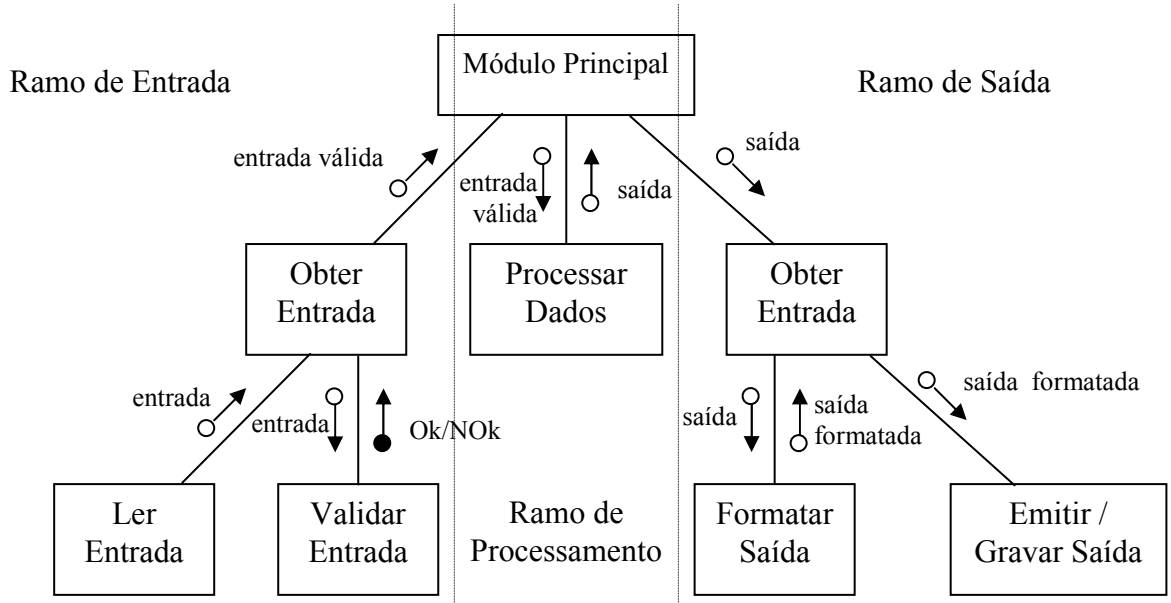


Figura 6.26 – Análise de Transformação.

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 6.27. 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 [4].

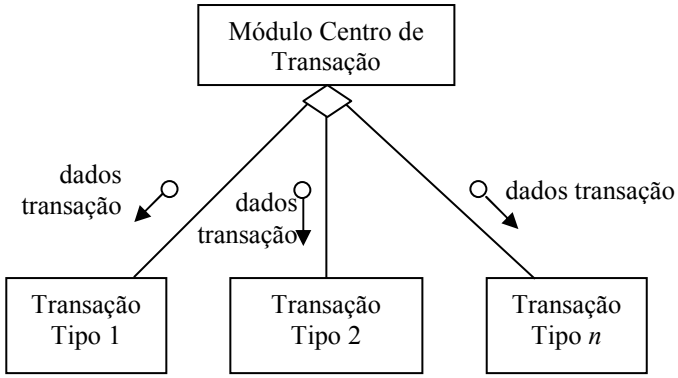


Figura 6.27 – Análise de Transação.

6.3.2 - Critérios de Qualidade de Projetos de Programa

O objetivo maior do projeto modular de programas é permitir que um sistema complexo seja dividido 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 desses 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 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 módulos, cada um deles 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 fluxos de controle, devemos fazê-la sem máscaras. Seja o exemplo da figura 6.28.

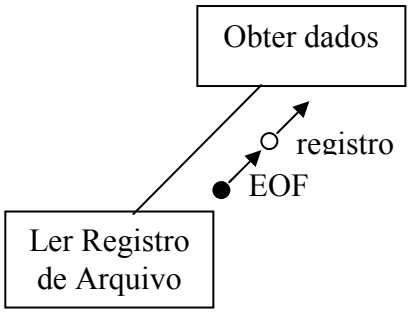


Figura 6.28 – Clareza na comunicação.

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

De maneira geral, não adianta melhorar dois desses 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 que chama não deve nunca enviar um controle ao módulo chamado**: isso significa que o módulo que chama está dizendo o que o módulo chamado deve fazer, caracterizando, portanto, que o módulo chamado não trata de uma única função.
- **Só utilizar fluxos de controle 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 variáveis globais**: Sempre que possível, utilizar variáveis locais.
- **É inadimissível 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 fluxos de controle, 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 6.29 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 a 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.

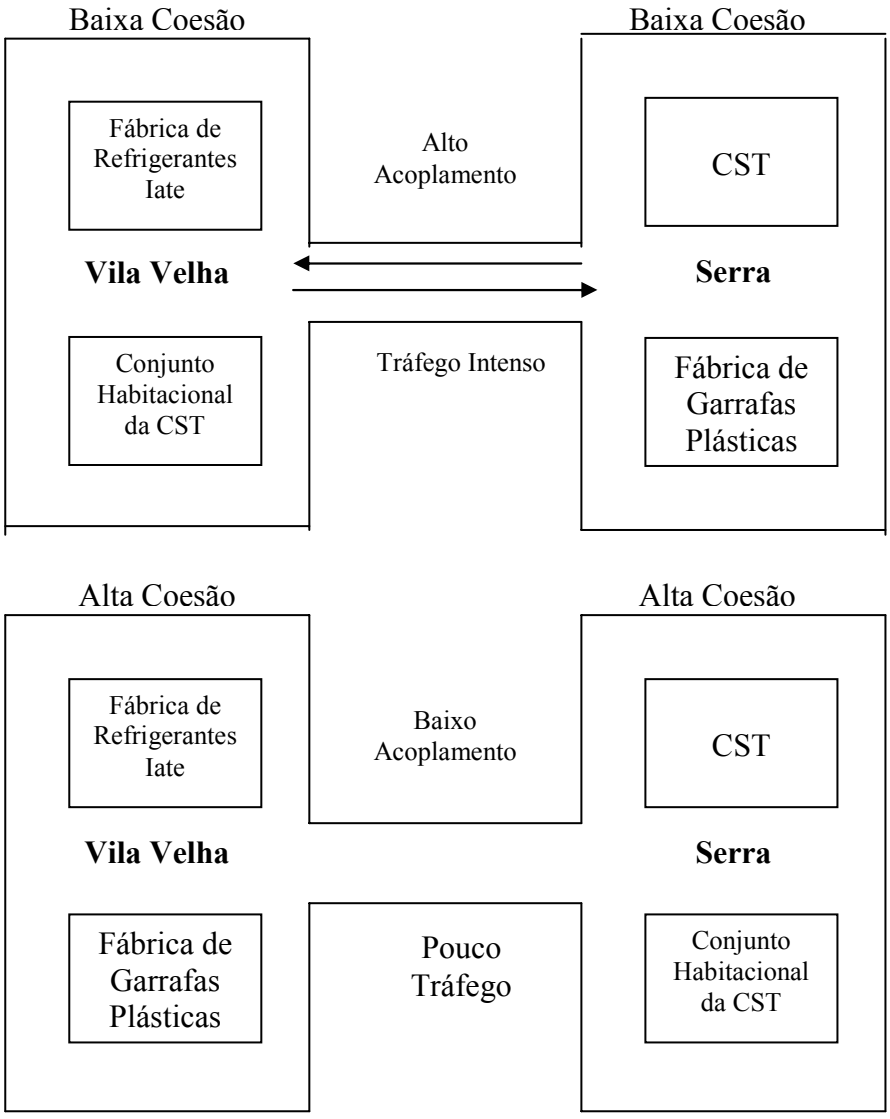


Figura 6.29 – Coesão e Acomplamento.

Referências Bibliográficas

1. R.S. Pressman, *Engenharia de Software*, Rio de Janeiro: McGraw Hill, 5ª edição, 2002.
2. I. Sommerville, *Engenharia de Software*, São Paulo: Addison-Wesley, 6ª edição, 2003.
3. S.L. Pfleeger, *Engenharia de Software: Teoria e Prática*, São Paulo: Prentice Hall, 2ª edição, 2004.
4. J. Martin, C. McClure. *Técnicas Estruturadas e CASE*. Makron Books, São Paulo, 1991.
5. C.M.S. Xavier, C. Portilho. *Projetando com Qualidade a Tecnologia de Sistemas de Informação*. Livros Técnicos e Científicos Editora, 1995.

7 – Implementação e Testes

Referências: [1] Cap. 17 e 18, [2] Cap. 20, [3] Cap. 7 e 8.

Uma vez projetado o sistema, é necessário escrever os programas que implementem esse projeto e testá-los.

7.1 - Implementação

Ainda que um projeto bem elaborado facilite sobremaneira a implementação, essa tarefa não é necessariamente fácil. Muitas vezes, os projetistas não conhecem em detalhes a plataforma de implementação e, portanto, não são capazes de (ou não desejam) chegar a um projeto algorítmico passível de implementação direta. Além disso, questões relacionadas à legibilidade, alterabilidade e reutilização têm de ser levadas em conta.

Deve-se considerar, ainda, que programadores, geralmente, trabalham em equipe, necessitando integrar, testar e alterar código produzido por outros. Assim, é muito importante que haja padrões organizacionais para a fase de implementação. Esses padrões devem ser seguidos por todos os programadores e devem estabelecer, dentre outros, padrões de nomes de variáveis, formato de cabeçalhos de programas e formato de comentários, recuos e espaçamento, de modo que o código e a documentação a ele associada sejam claros para quaisquer membros da organização.

Padrões para cabeçalho, por exemplo, podem informar o que o código (programa, módulo ou componente) faz, quem o escreveu, como ele se encaixa no projeto geral do sistema, quando foi escrito e revisado, apoios para teste, entrada e saída esperadas etc. Essas informações são de grande valia para a integração, testes, manutenção e reutilização [3].

Além dos comentários feitos no cabeçalho dos programas, comentários adicionais ao longo do código são também importantes, ajudando a compreender como o componente é implementado.

Por fim, o uso de nomes significativos para variáveis, indicando sua utilização e significado, é imprescindível, bem como o uso adequado de recuo e espaçamento entre linhas de código, que ajudam a visualizar a estrutura de controle do programa [3].

Além da documentação interna, escrita no próprio código, é importante que o código de um sistema possua também uma documentação externa, incluindo uma visão geral dos componentes do sistema, dos diversos grupos de componentes e da inter-relação entre eles [3].

Ainda que padrões sejam muito importantes, deve-se ressaltar que a correspondência entre os componentes do projeto e o código é fundamental, caracterizando-se como a mais importante questão a ser tratada. O projeto é o guia para a implementação, ainda que o programador deva ter certa flexibilidade para implementá-lo como código [3].

Como resultado de uma implementação bem-sucedida, as unidades de software devem ser codificadas e critérios de verificação das mesmas devem ser definidos.

7.2 - Testes

Uma vez implementado o código de uma aplicação, o mesmo deve ser testado para descobrir tantos defeitos quanto possível, antes da entrega do produto de software ao seu cliente.

Conforme discutido no capítulo 4, o teste é uma atividade de verificação e validação do software e consiste na análise dinâmica do mesmo, isto é, na execução do produto de software com o objetivo de verificar a presença de defeitos no produto e aumentar a confiança de que o mesmo está correto [4].

Vale ressaltar que, mesmo se um teste não detectar defeitos, isso não quer dizer necessariamente que o produto é um produto de boa qualidade. Muitas vezes, a atividade de teste empregada pode ter sido conduzida sem planejamento, sem critérios e sem uma sistemática bem definida, sendo, portanto, os testes de baixa qualidade [4].

Assim, o objetivo é projetar testes que potencialmente descubram diferentes classes de erros e fazê-lo com uma quantidade mínima de esforço [1]. Ainda que os testes não possam demonstrar a ausência de defeitos, como benefício secundário, podem indicar que as funções do software parecem estar funcionando de acordo com o especificado.

A idéia básica dos testes é que os defeitos podem se manifestar por meio de falhas observadas durante a execução do software. Essas falhas podem ser resultado de uma especificação errada ou falta de requisito, de um requisito impossível de implementar considerando o hardware e o software estabelecidos, o projeto pode conter defeitos ou o código pode estar errado. Assim, uma falha é o resultado de um ou mais defeitos [3].

São importantes princípios de testes a serem observados [1, 3]:

- Teste completo não é possível, ou seja, mesmo para sistemas de tamanho moderado, pode ser impossível executar todas as combinações de caminhos durante o teste.
- Teste envolve vários estágios. Geralmente, primeiro, cada módulo é testado isoladamente dos demais módulos do sistema (teste de unidade). À medida que os testes progridem, o foco se desloca para a integração dos módulos (teste de integração), até se chegar ao sistema como um todo (teste de sistema).
- Teste deve ser conduzido por terceiros. Os testes conduzidos por outras pessoas que não aquelas que produziram o código têm maior probabilidade de encontrar defeitos. O desenvolvedor que produziu o código pode estar muito envolvido com ele para poder detectar defeitos mais sutis.
- Testes devem ser planejados bem antes de serem realizados. Um plano de testes deve ser utilizado para guiar todas as atividades de teste e deve incluir objetivos do teste, abordando cada tipo (unidade, integração e sistema), como serão executados e quais critérios a serem utilizados para determinar quando o teste está completo. Uma vez que os testes estão relacionados aos requisitos dos clientes e usuários, o planejamento dos testes pode começar tão logo a especificação de requisitos tenha sido elaborada. À medida que o processo de desenvolvimento avança (análise, projeto e implementação), novos testes vão sendo planejados e incorporados ao plano de testes.

O processo de teste envolve quatro atividades principais [3, 4]:

- Planejamento de Testes: trata da definição das atividades de teste, das estimativas dos recursos necessários para realizá-las, dos objetivos, estratégias e técnicas de teste a serem adotadas e dos critérios para determinar quando uma atividade de teste está completa.
- Projeto de Casos de Testes: é a atividade chave para um teste bem-sucedido, ou seja, para se descobrir a maior quantidade de defeitos com o menor esforço possível. Os casos de teste devem ser cuidadosamente projetados e avaliados para tentar se obter um conjunto de casos de teste que seja representativo e envolva as várias possibilidades de exercício das funções do software (cobertura dos testes). Existe uma grande quantidade de técnicas de teste para apoiar os testadores a projetar casos de teste, oferecendo uma abordagem sistemática para o teste de software.
- Execução dos testes: consiste na execução dos casos de teste e registro de seus resultados.
- Avaliação dos resultados: detectadas falhas, os defeitos deverão ser procurados. Não detectadas falhas, deve-se fazer uma avaliação final da qualidade dos casos de teste e definir pelo encerramento ou não de uma atividade de teste.

7.2.1 – Técnicas de Teste

Para testar um módulo, é necessário definir um caso de teste, executar o módulo com os dados de entrada definidos por esse caso de teste e analisar a saída. Um teste é um conjunto limitado de casos de teste, definido a partir do objetivo do teste [3].

Diversas técnicas de teste têm sido propostas visando apoiar o projeto de casos de teste. Essas técnicas podem ser classificadas, segundo a origem das informações utilizadas para estabelecer os objetivos de teste, em, dentre outras categorias, técnicas funcional, estrutural ou baseadas em máquinas de estado [4].

Os testes funcionais ou caixa-preta utilizam as especificações (de requisitos, análise e projeto) para definir os objetivos do teste e, portanto, para guiar o projeto de casos de teste. O conhecimento sobre uma determinada implementação não é usado [4]. Assim, os testes são conduzidos na interface do software. Os testes caixa-preta são empregados para demonstrar que as funções do software estão operacionais, que a entrada é adequadamente aceita e a saída é corretamente produzida e que a integridade da informação externa (uma base de dados, por exemplo) é mantida [1].

Os testes estruturais ou caixa-branca estabelecem os objetivos do teste com base em uma determinada implementação, verificando detalhes do código. Caminhos lógicos internos são testados, definindo casos de testes que exercitem conjuntos específicos de condições ou laços [1].

Os testes baseados em máquinas de estado são projetados utilizando o conhecimento subjacente à estrutura de uma máquina de estados para determinar os objetivos do teste [4].

É importante ressaltar que técnicas de teste devem ser utilizadas de forma complementar, já que elas têm propósitos distintos e detectam categorias de erros distintas [4]. À primeira vista, pode parecer que realizando testes caixa branca rigorosos poderíamos chegar a programas corretos. Contudo, conforme anteriormente mencionado, isso não é prático, uma

vez que todas as combinações possíveis de caminhos e valores de variáveis teriam de ser exercitadas, o que é impossível. Isso não quer dizer, entretanto, que os testes caixa-branca não são úteis. Testes caixa-branca podem ser usados, por exemplo, para garantir que todos os caminhos independentes[1] de um módulo tenham sido exercitados pelo menos uma vez [1].

Há diversas técnicas de testes caixa-branca, cada uma delas procurando apoiar o projeto de casos de teste focando em algum ou vários aspectos da implementação. Dentre elas, podem ser citadas [1]:

- **Testes de estrutura de controle**: como o próprio nome diz, enfocam as estruturas de controle de um módulo, tais como comandos, condições e laços. Teste de condição é um tipo de teste de estrutura de controle que exercita as condições lógicas contidas em um módulo. Um teste de fluxo de dados, por sua vez, seleciona caminhos de teste tomando por base a localização das definições e dos usos das variáveis nos módulos. Testes de ciclo ou laço focalizam exclusivamente os laços (*loops*).
- **Teste de caminho básico**: define uma medida de complexidade lógica de um módulo e usa essa medida como guia para definir um conjunto básico de caminhos de execução.

Assim como há diversas técnicas de teste caixa-branca, o mesmo acontece em relação ao teste caixa-preta. Dentre as diversas técnicas de teste caixa-preta, podem ser citadas [1]:

- **Particionamento de equivalência**: divide o domínio de entrada de um módulo em classes de equivalência, a partir das quais casos de teste são derivados. A meta é minimizar o número de casos de teste, ficando apenas com um caso de teste para cada classe, uma vez que, a princípio, todos os elementos de uma mesma classe devem se comportar de maneira equivalente.
- **Análise de valor limite**: a prática mostra que um grande número de erros tende a ocorrer nas fronteiras do domínio de entrada de um módulo. Tendo isso em mente, a análise de valor limite leva à seleção de casos de teste que exercitem os valores limítrofes.

7.2.2 – Estratégias de Teste

O projeto efetivo de casos de teste é importante, mas não suficiente para o sucesso da atividade de testes. A estratégia, isto é, a série planejada de realização dos testes, é também crucial [1]. Basicamente, há três grandes fases de teste [4]:

- **Teste de Unidade**: tem por objetivo testar a menor unidade do projeto (um componente de software que não pode ser subdividido), procurando identificar erros de lógica e de implementação em cada módulo separadamente. No paradigma estruturado, a menor unidade refere-se a um procedimento ou função.
- **Teste de Integração**: visa a descobrir erros associados às interfaces entre os módulos quando esses são integrados para formar estrutura do produto de software.

[1] Um caminho independente é qualquer caminho ao longo de um módulo que introduz pelo menos um novo comando de processamento ou condição [1].

- **Teste de Sistema**: tem por objetivo identificar erros de funções (requisitos funcionais) e características de desempenho (requisito não funcional) que não estejam de acordo com as especificações.

Tomando por base essas fases, a atividade de teste pode ser estruturada de modo que, em cada fase, diferentes tipos de erros e aspectos do software sejam considerados [4]. Tipicamente, os primeiros testes focalizam componentes individuais e aplicam testes caixa-branca e caixa-preta para descobrir erros. Após os componentes individuais terem sido testados, eles precisam ser integrados, até se obter o sistema por inteiro. Na integração, o foco é o projeto e a arquitetura do sistema. Finalmente, uma série de testes de alto nível é executada quando o sistema estiver operacional, visando a descobrir erros nos requisitos [1,3].

No teste de unidade, faz-se necessário construir pequenos componentes para permitir testar os módulos individualmente, os ditos *drivers* e *stubs*. Um *driver* é um programa responsável pela ativação e coordenação do teste de uma unidade. Ele é responsável por receber os dados de teste fornecidos pelo testador, passar esses dados para a unidade sendo testada, obter os resultados produzidos por essa unidade e apresentá-los ao testador. Um *stub* é uma unidade que substitui, na hora do teste, uma outra unidade chamada pela unidade que está sendo testada. Em geral, um *stub* simula o comportamento da unidade chamada com o mínimo de computação ou manipulação de dados [4].

A abordagem de integração de módulos pode ter impacto na quantidade de *drivers* e *stubs* a ser construída. Sejam as seguintes abordagens:

- **Integração ascendente ou *bottom-up***: Nessa abordagem, primeiramente, cada módulo no nível inferior da hierarquia do sistema é testado individualmente. A seguir, são testados os módulos que chamam esses módulos previamente testados. Esse procedimento é repetido até que todos os módulos tenham sido testados [3]. Neste caso, apenas *drivers* são necessários. Seja o exemplo da figura 7.1. Usando a abordagem de integração ascendente, os módulos seriam testados da seguinte forma. Inicialmente, seriam testados os módulos do nível inferior (E, F e G). Para cada um desses testes, um *driver* teria de ser construído. Concluídos esses testes, passaríamos ao nível imediatamente acima, testando seus módulos (B, C e D) combinados com os módulos por eles chamados. Neste caso, testamos juntos B, E e F bem como C e G. Novamente, três *drivers* seriam necessários. Por fim, testaríamos todos os módulos juntos.

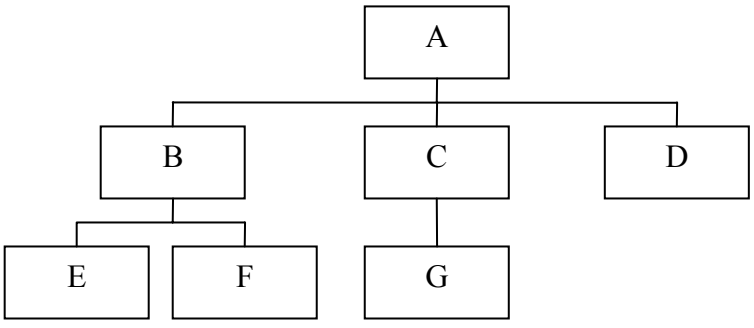


Figura 7.1 – Exemplo de uma hierarquia de módulos.

- **Integração descendente ou *top-down***: A abordagem, neste caso, é precisamente o contrário da anterior. Inicialmente, o nível superior (geralmente um módulo de controle) é testado sozinho. Em seguida, todos os módulos chamados por pelo

módulo testado são combinados e testados como uma grande unidade. Essa abordagem é repetida até que todos os módulos tenham sido incorporados [3]. Neste caso, apenas *stubs* são necessários. Tomando o exemplo da figura 7.1, o teste iniciaria pelo módulo A e três *stubs* (para B, C e D) seriam necessários. Na seqüência seriam testados juntos A, B, C e D, sendo necessários *stubs* para E, F e G. Por fim, o sistema inteiro seria testado.

Muitas outras abordagens, algumas usando as apresentadas anteriormente, podem ser adotadas, tal como a integração sanduíche [3], que considera uma camada alvo no meio da hierarquia e utiliza as abordagens ascendente e descendente, respectivamente para as camadas localizadas abaixo e acima da camada alvo. Outra possibilidade é testar individualmente cada módulo e só depois de testados individualmente integrá-los (teste *big-band*). Neste caso, tanto *drivers* quanto *stubs* têm de ser construídos para cada módulo, o que leva a muito mais codificação e problemas em potencial [3].

Uma vez integrados todos os módulos do sistema, parte-se para os testes de sistema, quando se busca observar se o software funciona conforme esperado pelo cliente. Por isso mesmo, muitas vezes, os testes de sistema são chamados de testes de validação. Os testes de sistema incluem diversos tipos de teste, realizados na seguinte ordem [3]:

- Teste funcional: verifica se o sistema integrado realiza as funções especificadas nos requisitos;
- Teste de desempenho: verifica se o sistema integrado atende os requisitos não funcionais do sistema (eficiência, segurança, confiabilidade etc);
- Teste de aceitação: os testes funcional e de desempenho são ainda realizados por desenvolvedores, entretanto é necessário que o sistema seja testado pelos clientes. No teste de aceitação, os clientes testam o sistema a fim de garantir que o mesmo satisfaz suas necessidades. Vale destacar que o que foi especificado pelos desenvolvedores pode ser diferente do que queria o cliente. Assim, o teste de aceitação assegura que o sistema solicitado é o que foi construído.
- Teste de instalação: algumas vezes o teste de aceitação é feito no ambiente real de funcionamento, outras não. Quando o teste de aceitação for feito em um ambiente de teste diferente do local em que será instalado, é necessário realizar testes de instalação.

Referências

1. R.S. Pressman, *Engenharia de Software*, Rio de Janeiro: McGraw Hill, 5ª edição, 2002.
2. I. Sommerville, *Engenharia de Software*, São Paulo: Addison-Wesley, 6ª edição, 2003.
3. S.L. Pfleeger, *Engenharia de Software: Teoria e Prática*, São Paulo: Prentice Hall, 2ª edição, 2004.
4. J.C. Maldonado, S.C.P.F. Fabbri, "Teste de Software". In: *Qualidade de Software: Teoria e Prática*, Eds. A.R.C. Rocha, J.C. Maldonado, K. Weber, Prentice Hall, 2001.

8 – Entrega e Manutenção

Referências: [3] Cap. 10 e 11.

Concluídos os testes, sistema aceito e instalado, estamos chegando ao fim do processo de desenvolvimento de software. A entrega é a última etapa desse processo. Uma vez entregue, o sistema passa a estar em operação e eventuais mudanças, sejam de caráter corretivo, sejam de caráter de evolução, caracterizam-se como uma manutenção.

8.1 - Entrega

A entrega não é meramente uma formalidade. No momento em que o sistema é instalado no local de operação e devidamente aceito, é necessário, ainda, ajudar os usuários a entenderem e a se sentirem mais familiarizados com o sistema. Neste momento, duas questões são cruciais para uma transferência bem-sucedida: treinamento e documentação [3].

A operação do sistema é extremamente dependente de pessoal com conhecimento e qualificação. Portanto, é essencial que o treinamento de pessoal seja realizado para que os usuários e operadores possam operar o sistema adequadamente.

A documentação que acompanha o sistema também tem papel crucial na entrega, afinal ela será utilizada como material de referência para a solução de problemas ou como informações adicionais. Essa documentação inclui, dentre outros, manuais do usuário e do operador, guia geral do sistema, tutoriais, ajuda (*help*), preferencialmente on-line e guias de referência rápida [3].

8.2 - Manutenção

O desenvolvimento de um sistema termina quando o produto é entregue para o cliente e entra em operação. A partir daí, deve-se garantir que o sistema continuará a ser útil e atendendo às necessidades do usuário, o que pode demandar alterações no mesmo. Começa, então, a fase de manutenção [4].

Há muitas causas para a manutenção, dentre elas [4]: falhas no processamento devido a erros no software, falhas de desempenho, alterações no ambiente de dados, alterações no ambiente de processamento, necessidade de modificações em funções existentes e necessidade de inclusão de novas capacidades.

Ao contrário do que podemos pensar, a manutenção não é uma atividade trivial nem de pouca relevância. Ela é uma atividade importantíssima e de intensa necessidade de conhecimento. O mantenedor precisa conhecer o sistema, o domínio de aplicação, os requisitos do sistema, a organização que utiliza o mesmo, práticas de engenharia de software passadas e atuais, a arquitetura do sistema, algoritmos usados etc.

O processo de manutenção é semelhante, mas não igual, ao processo de desenvolvimento e pode envolver atividades de levantamento de requisitos, análise, projeto, implementação e testes, agora no contexto de um software existente. Essa semelhança pode ser maior ou menor, dependendo do tipo de manutenção a ser realizada.

Pfleeger [3] aponta os seguintes tipos de manutenção:

- **Manutenção corretiva**: trata de problemas decorrentes de defeitos. À medida que falhas ocorrem, elas são relatadas à equipe de manutenção, que se encarrega de encontrar o defeito que causou a falha e faz as correções (nos requisitos, análise, projeto ou implementação), conforme o necessário. Esse reparo inicial pode ser temporário, visando manter o sistema funcionando. Quando esse for o caso, mudanças mais complexas podem ser implementadas posteriormente.
- **Manutenção adaptativa**: às vezes, uma mudança no ambiente do sistema, incluindo hardware e software de apoio, pode implicar em uma necessidade de adaptação.
- **Manutenção perfectiva**: consiste em realizar mudanças para melhorar algum aspecto do sistema, mesmo quando nenhuma das mudanças for conseqüência de defeitos. Isso inclui a adição de novas capacidades bem como ampliações gerais.
- **Manutenção preventiva**: consiste em realizar mudanças a fim de prevenir falhas. Geralmente ocorre quando um mantenedor descobre um defeito que ainda não causou falha e decide corrigi-lo antes que ele gere uma falha.

Referências

3. S.L. Pfleeger, *Engenharia de Software: Teoria e Prática*, São Paulo: Prentice Hall, 2ª edição, 2004.
4. R. Sanches, "Processo de Manutenção". In: *Qualidade de Software: Teoria e Prática*, Eds. A.R.C. Rocha, J.C. Maldonado, K. Weber, Prentice Hall, 2001.